Hybrid Hardware/Software Architectures for Network Packet Processing in Security Applications

Dissertation

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von
Andreas Christoph Kurt Fießler

Präsidentin der Humboldt-Universität zu Berlin
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät
Prof. Dr. Elmar Kulke

Gutachter:
1. Prof. Dr. Björn Scheuermann
2. Dr. Andrew W. Moore
3. Prof. Dr. Georg Carle

Tag der mündlichen Prüfung: 8. Mai 2019

Hybrid Hardware/Software Architectures for Network Packet Processing in Security Applications

---

Andreas Christoph Kurt Fießler

Abstract

The amount of data processed by computer networks is growing faster than processing and memory performance. Network devices like switches, bridges, routers, and firewalls are subject to a continuous development to keep up with ever-rising requirements. As the overhead of software network processing already became the performance-limiting factor for a variety of applications, also former software functions are shifted towards dedicated network processing hardware. Although such application-specific circuits allow fast, parallel, and low latency processing, they require expensive and time-consuming development with minimal possibilities for adaptions. Security can also be a major concern, as these circuits are virtually a black box to everyone except for the manufacturer. Moreover, the highly parallel processing capabilities of specialized hardware are not necessarily an advantage for all kinds of tasks in network processing, where sometimes a classical CPU is better suited.

This work introduces and evaluates concepts for building hybrid hardware-software-systems that exploit the advantages of both hardware and software approaches in order to achieve performant, flexible, and versatile network processing systems. We focus on the use case of firewalls and the underlying network packet classification problem. The approaches are evaluated on standard software systems, extended by a plug-in card for hardware acceleration based on programmable hardware circuits (FPGAs) to provide full control and flexibility. The hybrid approach further allows the use of logic-level optimized, resource efficient hardware filtering circuits on the FPGA. One key achievement of this work is the identification and mitigation of challenges inherent when a hybrid combination of multiple packet classification circuits with different characteristics is used. We introduce approaches to reduce redundant classification effort to a minimum, like re-usage of intermediate classification results and determination of dependencies by *header space analysis*. In addition, for some further challenges in hardware based packet classification like filtering circuits with dynamic updates and fast hash functions for lookups, we describe feasibility and optimizations. At last, the hybrid approach is evaluated using a standard SDN switch instead of the FPGA accelerator to prove portability. Again, significant performance gains could be achieved.

Zusammenfassung

Die Menge an in Computernetzwerken verarbeiteten Daten steigt stetig und schneller als verfügbare Rechenleistung und Speicher. Dies stellt Netzwerkgeräte wie Switches, Bridges, Router und Firewalls vor Herausfordungen, die eine kontinuierliche Weiterentwicklung erfordern. Die Performance der verbreiteten, CPU/softwarebasierten Ansätze für die Implementierung dieser Aufgaben ist durch den inhärenten Overhead in der sequentiellen Datenverarbeitung limitiert, weshalb solche Funktionalitäten vermehrt auf dedizierten Hardwarebausteinen realisiert werden. Diese bieten eine schnelle, parallele Verarbeitung mit niedriger Latenz, sind allerdings aufwendiger in der Entwicklung und weniger flexibel. Nicht jede Anwendung kann zudem für hochgradig parallele Verarbeitung optimiert werden.

Diese Arbeit befasst sich mit hybriden Ansätzen, um eine bessere Ausnutzung der jeweiligen Stärken von Soft- und Hardwaresystemen für Netzwerkanwendungen zu ermöglichen, mit Schwerpunkt auf der Paketklassifikation. Es wird eine Firewall realisiert, die sowohl Flexibilität und Analysetiefe einer Software-Firewall als auch Durchsatz und Latenz einer Hardware-Firewall erreicht. Der Ansatz wird auf einem Standard-Rechnersystem, welches für die Hardware-Klassifikation mit einem rekonfigurierbaren Logikbaustein (FPGA) ergänzt wird, evaluiert. Das hybride Konzept ermöglicht dabei die Verwendung von hochgradig optimierten Filterungsschaltkreisen, welche den Ressourcenverbrauch auf dem FPGA minimieren. Eine wesentliche Herausforderung einer hybriden Firewall mit verschiedenartigen Klassifikationssystemen ist die Identifikation von Abhängigkeiten im Regelsatz. Es werden Ansätze vorgestellt, welche den redundanten Klassifikationsaufwand auf ein Minimum reduzieren, wie etwa die Wiederverwendung von Teilergebnissen der hybriden Klassifikatoren oder eine exakte Abhängigkeitsanalyse mittels *Header Space Analysis*. Für weitere Problemstellungen im Bereich der hardwarebasierten Paketklassifikation, wie dynamisch konfigurierbare Filterungsschaltkreise und schnelle, sichere Hashfunktionen für Lookups, werden Machbarkeit und Optimierungen evaluiert. Der hybride Ansatz wird im Weiteren auf ein System mit einer SDN-Komponente statt einer FPGA-Erweiterung übertragen. Auch hiermit können signifikante Performancegewinne erreicht werden.

Acknowledgement

In this short preface, I would like to take the opportunity to thank all the people who contributed to this thesis, be it through feedback, discussions or just accompanying me during these years. The course of this work, in particular in the beginning, was far from being straightforward. A major part of my decision to follow this path should be attributed to Björn Scheuermann, and his initial offer to be the advisor of the thesis. Björn, thank you for your continuous support, contributions, feedback and responsiveness during these years, all of which were at a level that I consider way above what is usual. During my research, I had to privilege to spend several months at the Computer Lab of the University of Cambridge as a member of Andrew W. Moore's group. Andrew, thank you for this experience, your feedback for my papers and this thesis, and the opportunities that came with it. This stay abroad was supported with a generous grant awarded by the Competence Center for Applied Security Technology, Darmstadt. I would further like to thank the third reviewer, Georg Carle, and the committee members Ulf Leser, Jens-Peter Redlich and Holger Döbler for their constructive feedback and suggestions.

A significant amount of research and publication was accomplished together with Sven, who I first met during the research project HARDFIRE as a member of the Humbold University and who later became a colleague of mine. I really enjoyed all the days, nights and weekends we spent with discussions, building setups and facing submission deadlines. Considerable thanks also goes to Claas and Daniel for their valuable, numerous contributions and their efforts in our joint work.

The opportunity to pursue a PhD while also having a regular employment should also not be taken for granted, especially with the countless great colleagues that make this place—the genua GmbH—such an enjoyable place to work at. I would particularly like to thank Alexander for supporting my PhD from the beginning, and also Johanna for often enough pushing me out of my comfort zone and cheering me up. Additionally, the department of computer science at the Humbold University of Berlin with all its members has always been a great place for valuable feedback on ideas, papers and presentations.

My ambitions for this path, starting with the first day at the university, would have been significantly more difficult to follow without the support of my family. Thank you, Christiane and Bernd, for your caring encouragement. Finally, I would

like to thank my partner Vanessa for accompanying me all the time, never letting me down despite all the challenges, and being someone I can count on.

Andreas Fießler
Munich
November, 2018

Contents

1 Introduction

1.1 Motivation and Outline

Since the widespread emergence of computer networks and the Internet in particular, the continuous, exponential growth of transmitted data has been a reliable observation. As more and more devices are connected, services are moved to cloud services and requirements are rising. This growth can be expected to continue. According to Cisco, the annual global *Internet Protocol* (IP) traffic was 1.2 ZB in 2016 and will reach 3.3 ZB per year by 2021 [44]. More tangible, this means 16 GB of traffic per capita in 2016 and 35 GB in 2021. An often cited law by GERRY BUTTER postulated an even higher growth—doubling every nine months—in optical networks [124]. Regardless of the exact numbers, this significant rise increases the requirements for all devices handling this traffic. This includes switches, routers, gateways, firewalls, and *intrusion detection systems* (IDSs) to name but a few. On the physical layer, new technologies and standards are regularly introduced to increase bandwidth and throughput. The de-facto standard for local area networks, Ethernet, defines standards for link speeds of up to 100 Gbit/s (100GBASE), while 400 Gbit/s is currently developed [110]. That alone, however, is only one part of the problem as these network devices need to perform more or less complex tasks with the actual packet.

A common operation that is done by almost all of the network devices is *network packet classification*, which compares certain properties of a packet against a set of rules to determine an action to perform. We will refer to systems dedicated to this task as *packet classification systems*. While this task can be accomplished on systems based on either software or dedicated hardware, software systems hardly meet line-speed requirements due to the inherent processing overhead [49]. Consequently, systems like switches and routers can commonly be found equipped with *application-specific integrated circuits* (ASICs). Such ASICs often include configuration registers and memory which allow a limited, dynamic configuration. ASICs dedicated for network operations are also called *network processing units* (NPUs). Switches dedicated for *software-defined networking* (SDN) also mostly rely on ASICs for the data plane when higher throughput is needed. For security systems like firewalls and IDS however, evolving attacks lead to higher requirements exceeding simple rules that only classify based on, e.g., port num-

bers and addresses [26]. Examples for such complex analysis include stateful packet filtering or statistical analysis and go up to application-layer functions like proxying and intercepting protocols and malware detection [107]. These sort of inspections do not scale well on dedicated hardware, especially when algorithms need to be updated frequently [20]. In summary, hardware-based devices provide unrivaled high throughput and low latency as long as the tasks are stateless and comparatively simple. Network processing can also be implemented on *field-programmable gate arrays* (FPGAs) [20, 25, 49, 59], which can, to some extent, exploit advantages of hardware-based approaches, while the hardware design can—in contrast to ASICs—easily be replaced. Software-based systems are limited in throughput, but benefit from a fast general purpose *central processing unit* (CPU) and virtually unlimited memory for advanced tasks. Still, also software systems need to perform a significant amount of simple classification work in the first place. This motivates to evaluate whether a combination of different packet classification systems—each working as a compartment in a hybrid setup—is feasible. The combination of a software- and a hardware-based system is particularly interesting, as both bring diverse characteristics.

Hence, we formulate the first research question:

> How can we realize a combination of two packet classification systems so that each compartment processes the tasks it is best suited for in order to achieve high throughput, flexibility and comprehensive analysis capabilities?

FPGA-based network packet processing in particular bears certain challenges that are inherent to the technology. However, hardware-centric approaches promise great potential for better performance. This leads to the second research question:

> Which network packet processing tasks are desirable and feasible for FPGA implementation and what implementation approaches and optimizations are possible?

Hybrid systems may also be implemented with three or more subsystems. Nevertheless, this case is not within the scope of this thesis, apart from identifying possible starting points. Apart from this, for practical applications, the portability of the approaches is of interest. This thesis aims at a concept that is not only applicable to a certain test setup. Moreover, modifications on standard components like the software firewall used in the hybrid system should be avoided. The third research question is therefore:

How can the hybrid classification approach be applied to other types of packet classification systems if one goal is to avoid modifications to the components themselves?

1.2 Results and Contributions

The key achievements and contributions of this thesis are the following:

- We demonstrate the feasibility of a hybrid firewall system. This involves the combination of different packet classification systems with different characteristics with regard to matching capabilities, performance, rule set capacity, update effort, and costs. This is evaluated with a combination of a standard software firewall and an FPGA-based plug-in card, the latter using a logic-level optimized filtering approach. The proposed hybrid system ensures the correct classification result given by the rule set under all conditions.

- Using logic-level optimized hardware classification circuits on FPGAs allows resource-efficient implementations, but implicates slow update cycles. The proposed hybrid firewall can bridge update delays by temporarily reverting to the other packet classification compartment without interrupting the network traffic until the hardware packet classification circuit is regenerated.

- We show that our hybrid approach can mitigate the drawbacks of hardware- and software-based firewall systems, respectively. This results in a firewall system that profits from the expressiveness and versatility of a software firewall solution as well as the speed and latency of a hardware classification system. From an economic perspective, the approach allows using cheaper standard components compared to, e.g., developing a full-fledged classification and analysis on a hardware firewall.

- We develop and describe a protocol for managing hybrid firewall systems. The protocol is prototypically implemented in a tool that serves as an abstraction layer between administration level and the hybrid subsystems. For this purpose, it keeps track of the rule set and updates to the rule set, providing the corresponding configuration for both of the hybrid subsystems. It is equipped with analysis algorithms in order to calculate an efficient solution in terms of redundant classification effort. This way, significant performance gains can be achieved by the hybrid firewall. Only an interface comparable to a standard software firewall needs to be exposed to the user.

- Compared to logic-level optimized hardware packet classification circuits, hardware packet classification circuits with support for dynamic updates, i.e., generic circuits with on-the-fly configuration of the rule set, require significantly more resources and power. We exploit the fact that in practise, large parts of firewall rule sets are rarely changed and propose and evaluate a concept to effectively combine static, optimized, and dynamic circuits on a single FPGA. This approach can reduce power and resource consumption of the overall classification circuit.

- One key component for further extending network packet processing on FPGAs is the key-value-lookup, which typically relies on fast hash functions. We analyse efficient hash algorithms on FPGAs. We show that many commonly used hash functions are in fact not well-suited for hardware-implementation with regard to latency. For certain types of hash table applications typically used in network systems, we show that cryptographic security properties of hash functions are important for a proper use of the hash result. We provide a recommendation for such hash table use cases on FPGAs.

- We demonstrate that the developed hybrid approach can also be applied to other constellations, i.e., a standard software firewall in combination with an SDN switch. This exploits the SDN switching hardware as a fast bypass mechanism for simple classification decisions. Complex decisions, which would normally be diverted to an SDN controller via a slow control link, are offloaded to the software firewall using high-speed data links. This concept allows the integration of full-fledged firewalls in SDNs without relinquishing the high throughput and low latency of dedicated SDN networking hardware.

1.3 Thesis Structure

The remainder of this thesis continues with a formal introduction of the network packet classification problem and the discussion of related work in Chapter 2. Different implementation approaches used for tackling this problem are given with examples and their characteristics. As a major part of this thesis focusses on FPGA-based network packet classification, the related work on this topic and the evaluation system is introduced. In Chapter 3, the hybrid software/FPGA firewall HyPaFilter is introduced. Several strategies for distributing rules and traffic in hybrid systems are described and evaluated with regard to the packet classification performance and update latency. This work is extended in Chapter 4,

where HyPaFilter is augmented by an optimized packet routing decision. The chapter finishes with a comprehensive evaluation and comparison. In Chapter 5, we analyse certain subtopics regarding FPGA-based network packet processing. For two of those, i.e., dynamically updatable classification circuits and hash tables, implementation approaches and recommendations are given. Chapter 6 is dedicated to hybrid firewalls with SDN components. The techniques elaborated with HyPaFilter are applied to software firewall in conjunction with an SDN switch. Chapter 7 summarizes this thesis and gives a brief outlook on further research opportunities.

2 Network Packet Processing

The vast majority of computer networks implement packet-based protocols for data transfer [90]. Therefore, any device attached to such networks must be able to process and handle network packets. The complexity of this task strongly depends on the desired purpose, i.e., the protocol layer on which it has to operate on and the nature of the protocols running on these layers. In this chapter, we focus on one central challenge for networking devices: the packet classification problem. Furthermore, different types of systems used for network packet processing are introduced and compared. Based on this, we propose several types of packet processing approaches to be used for the following work.

2.1 Packet Classification

Analyzing packets based on header fields of the *Internet Protocol* (IP) and *Transmission Control Protocol* (TCP) or *User Datagram Protocol* (UDP) headers is commonly described as the packet classification problem. Due to its importance, it is a widely discussed topic in the research community [35, 88]. For this work, we use the following definition of a header $\mathcal{H}$. Let

$$\mathcal{H} = (H_1 \in D_1, \ldots, H_K \in D_K)$$

be a tuple of *header field values* with D_j being the *domain* of the jth header field. In this work, D_j is a range of non-negative integers, so header fields like IP addresses or TCP/UDP port numbers can be described. The set of all possible header values is

$$\mathcal{U} = D_1 \times \ldots \times D_K$$

so that $\mathcal{H} \in \mathcal{U}$. Devices relying on network packet classification will typically determine the result within a set of multiple rules. This rule set itself consists of a prioritized list of rules R_i, so we define a rule set as

$$\mathcal{R} = \langle R_1, \ldots, R_N \rangle.$$

```
-d 192.0.2.1/32 -p tcp --dport 80 -j ACCEPT                    # Rule R1
#-------------------------------------------------------------------------------
-s 192.0.2.0/24 -p udp -j ACCEPT                               # Rule R2
#-------------------------------------------------------------------------------
-d 192.0.2.0/32 -p tcp --dport 23 -j DROP                      # Rule R3
#-------------------------------------------------------------------------------
-d 192.0.2.1/32 -p icmp -j ACCEPT                              # Rule R4
```

Listing 2.1: Example rule set $\mathcal{R}$ in `iptables` syntax.

The rules, in turn, include one or more *checks* in order to determine the match. Therefore, each rule R_i incorporates K checks $C_i^j : D_j \to \{\text{true}, \text{false}\}$, so that

$$R_i = C_i^1 \wedge \ldots \wedge C_i^{K_i}.$$

The matching of a rule R_i with regard to the header tuple $\mathcal{H}$ is denoted by $R_i(\mathcal{H})$. It is true if $C_i^j(H_j)$ is true for all $j \in \{1, \ldots, K_i\}$. The common *first rule match* mode is used for this thesis—this means in a given rule set $\mathcal{R}$, out of all matching rules $R_i(\mathcal{H})$, the one with the lowest index i is decisive, i.e., is the one with the highest priority [1]. Therefore, the packet classification problem can be described as for a given $\mathcal{R}$ and $\mathcal{H}$, finding the rule with the lowest index i^* such that rule R_{i^*} matches $\mathcal{H}$. Each rule is associated with an *action*, e.g., ACCEPT, REJECT, or DROP, which can then be applied to the packet if a match is found.

Listing 2.1 shows a toy example of a rule set in the `netfilter/iptables`-firewall syntax style which will be used throughout this work. This example rule set would allow TCP traffic to address `192.0.2.1` and port `80` (usually used for the *Hypertext Transfer Protocol* (HTTP)), allow UDP traffic from source network `192.0.2.0/24`, forbid TCP connections to `192.0.2.0/24` and port 23 (usually the protocol *Telnet*) and allow *Internet Control Message Protocol* (ICMP) packets to the address `192.0.2.1`.

2.2 Extended Packet Classification Problem

The challenge of the packet classification problem can be extended by introducing two rule sub-types, with regard to the complexity of the matching criterion. The rules described in Section 2.1 contain checks C, which will be called simple checks. In practise, these criteria are not sufficient for a firewall setup [26], where more sophisticated techniques like connection tracking, probability-based matching or deep packet inspection are desired. The firewall still needs to be able to classify

[1] This stands in contrast to the *last rule match*, which is used by, e.g., pf [119]. Here, the rule with the highest index i is decisive, unless a rule with a lower index is tagged with a certain keyword that forces ending the matching process. It is possible to convert rule sets of both types to an equivalent rule set of the other type [94].

```
-d 1.2.3.4/32 -p tcp --dport 80        -j ACCEPT                          # Rule R1
#-------------------------------------------------------------------------------
-s 1.2.3.0/24 -p tcp                  -j ACCEPT                          # Rule R2
#-------------------------------------------------------------------------------
-d 1.2.3.5/32 -p udp --dport 3306 -m string                              # Rule R3
   --string "SELECT" --algo bm        -j DROP                            #
#-------------------------------------------------------------------------------
-d 1.2.3.4/32 -p tcp --dport 443       -j ACCEPT                          # Rule R4
#-------------------------------------------------------------------------------
-d 1.2.3.6/32 -p udp --dport 53 -m string                                # Rule R5
   --hex-string "|11|2|00|" --algo bm -j DROP                            #
#-------------------------------------------------------------------------------
-d 1.2.3.6/32 -p udp --dport 53        -j ACCEPT                          # Rule R6
```

Listing 2.2: Example rule set $\mathcal{R}$ in `iptables` syntax.

the packet against simple criteria as well. A rule in a firewall supporting complex checks can be described as:

$$R_i = C_i^1 \wedge \ldots \wedge C_i^{K_i} \wedge E_i,$$

where E_i are the summarized complex checks for each rule, specific for each rule R_i. A rule R_i is called a *simple rule* iff it does not contain any complex matching criteria. If complex checks are present, R_i is called a *complex rule*. For a given rule set $\mathcal{R}$, we denote the sublist of simple rules by $\mathcal{R}_S$, and the sublist of complex rules by $\mathcal{R}_C$. Therefore, $\mathcal{R}_S \cup \mathcal{R}_C = \mathcal{R}$ and $\mathcal{R}_S \cap \mathcal{R}_C = \emptyset$. For every simple rule $R_i \in \mathcal{R}$, let $\mathcal{R}_S(i)$ be the index of rule R_i in $\mathcal{R}_S$.

Listing 2.2 shows an example rule set $\mathcal{R} = \langle R_1, R_2, R_3, R_4, R_5, R_6 \rangle$ that consists of the simple sublist $\mathcal{R}_S = \langle R_1, R_2, R_4, R_6 \rangle$ and the complex sublist $\mathcal{R}_C = \langle R_3, R_5 \rangle$. Each rule specifies simple checks on source or destination addresses, the transport layer protocol, or the destination port (indicated through the flags `-s`, `-d`, `-p`, or `--dport`, respectively). Rules R_3 and R_5 are complex, as they define string matches on the packets' payload. It can be seen that the simple rules R_4 and R_6 are located at the indices 3 and 4 in $\mathcal{R}_S$, hence $\mathcal{R}_S(4) = 3$ and $\mathcal{R}_S(6) = 4$.

2.3 Classification Systems

The processing of packets in networking devices is typically structured in protocol layers. For the lowest layer, the computational challenges arise mostly due to physical effects while encoding and decoding the frame to and from the desired medium [74]. For higher layers, the processing complexity depends on the protocols that are used. An example for a demanding task in this context are application layer protocols running on top of, e.g., TCP/IP, where features for these protocols like packet reordering, fragmentation, and error control must be considered first before the application protocol can be processed. Typical network security devices like firewalls focus on the protocols running at the layers above

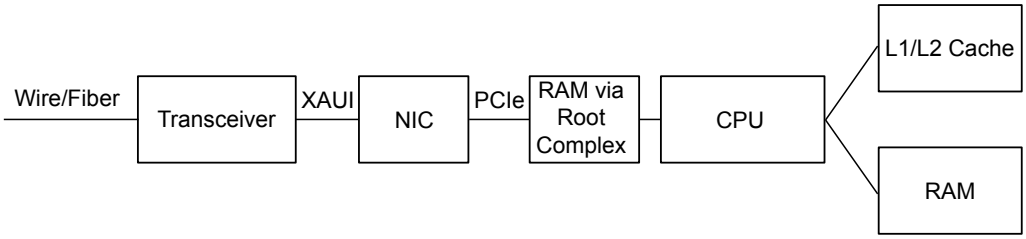


Fig. 2.1: Typical entities involved for packet processing on a standard software system.

the lowest-level ones responsible for physical and medium access. For example, those protocols include IPv4, IPv6, ICMP, TCP, and UDP, to name but a few. It is also becoming increasingly important to implement capabilities to analyse the payload contained in, e.g., TCP and UDP packets to prevent more sophisticated attacks [84].

2.3.1 Software-based Systems

Software-based network packet processing can be implemented with relatively small effort, as common operating systems often include necessary drivers, protocol stacks, and *application programming interfaces* (APIs). Consequently, a large number of software packet classification systems like firewalls have been developed. The firewall `netfilter` is a popular example of a pure software packet classification system. An interesting measure for software-based classification systems is the processing overhead. In a standard software-based setup based on *commercial off-the-shelf* (COTS) hardware running an operating system, data needs to be transferred from the *network interface card* (NIC) via several buses and switches to the hosts memory, where a software application can fetch the data and execute operations on the CPU. Figure 2.1 visualizes this path. For minimum sized Ethernet frames of 64 B (plus preamble, *start frame delimiter* (SFD) and *interframe gap* (IFG)), the maximum packet rate for a 40 Gbps link is about 60 Mpps. This means in the worst case, about every 17 ns one packet must be processed, including all the inherent overhead due to the sketched processing path which comes on top on a packet-by-packet basis. For comparison, this equals the required time for about 50 clock cycles on a 3 GHz single core CPU. This overhead is a significant limitation for the achievable throughput and latency of a software-based networking device. Such COTS software systems are therefore not suitable for line-speed requirements [49].

2.3.2 Optimizations for Software-based Systems

Several techniques have been introduced to improve the handling of network traffic on standard COTS systems. These approaches can be categorized into purely algorithmic optimizations of the classification process and general improvements of the processing flow.

Algorithmic Optimizations for Software Packet Classification

Optimizing software classification algorithms has been of major interest in the research community [10, 36, 6, 87]. Most common software classification systems like `netfilter` [127] and `pf` [119] use a plain linear search algorithm to find the first matching rule. Other approaches exist in the form of decision tree algorithms [36, 82], bit vector searches [10, 54], or hash maps [87]. These approaches aim at improving the search algorithm, which often results in a better classification performance [35, 38]. In addition, rule set transformation techniques can be applied to the rule set itself in order to achieve an equivalent rule set $\mathcal{R}'$ that can be traversed faster. The effectiveness of, e.g., rule set minimization or decision tree structures has been shown in [38, 56].

Processing Optimizations

The probably most popular optimization for network packet processing on standard systems is the *Data Plane Development Kit* (DPDK) [126]. The DPDK achieves significant performance gains in comparison to packet processing using standard operating system functions. In particular, DPDK uses optimized drivers, avoids context switches, and uses polling instead of interrupts [29, 32]. Plain performance tests show that DPDK-systems can achieve line rate for packet sizes of 128 Byte or larger [45]. The DPDK does not, however, natively implement layer 3-functions such as IPsec or firewalling. Reverting traffic to a standard software firewall would therefore require to waive some of the key factors for its performance. The firewall project *pfSense* incorporates the features of DPDK and has the goal to achieve a packet filtering throughput of 10 Gbps [63]. Another popular framework is *netmap*, which claims to achieve similar data rates by using techniques like, amongst others, memory-mapping packet buffers [75]. A different approach is used by the to-be standard firewall in Linux, the *extended Berkeley Packet Filter* (eBPF), where *just-in-time* (JIT) compilation is used to translate firewall policies. The compiled sources are then executed in eBPF virtual machines. Recent results show that eBPF can achieve significant performance

gains for packet filtering of up to 45% compared to using the standard `netfilter` framework [78]. However, the performance is still insufficient for achieving line rate in complex use cases with high-speed interfaces (i.e., 40 Gbit/s and above). Moreover, the measurements indicate that using eBPF can increase the packet processing latency.

2.3.3 Hardware-based Systems

As noted before, classification systems can be implemented as a software application or in dedicated hardware components. While the advantage of software systems lies in the flexibility and availability, hardware-based systems can operate faster and with lower latency. On hardware, most approaches aim at exploiting the possible parallelism in order to achieve high classification performance.

TCAMs

Simple classification tasks can be accomplished by *ternary content-addressable memory* (TCAM)-like structures, in which the entire rule set is compared against the packet's header data in a small, constant number of clock cycles [73]. This is achieved by storing all rules in a generic array of registers and distributing the header data for logic comparison. This makes TCAMs fast and deterministic. Hardware components—sometimes called *network search engine* (NSE) [102]—for simple high-speed classification tasks in, e.g., routers or switches are often based on TCAMs [86]. However, due to the massively parallel and generic layout, their implementation occupies a lot of circuit resources [73]. With increasing storage width and depth, TCAMs are therefore increasingly expensive and suffer from high power dissipation.

A further challenge is the representation of rules in TCAMs and comparable structures: they can only match on each bit position for 1, 0, or *don't care*. Hence, there is no native way to map negation (i.e., rules matching $C_i^j(H_j) = false$ for at least one j) or arbitrary range tests, like port ranges, into a TCAM [19, 93]. This can lead to possibly significantly reduced rule space due to necessary rule transformations [76]. To illustrate this problem, take the following range (decimal and binary) of IPv4 addresses:

```
192.168.1.1:  11000000.10101000.00000001.00000001 to
192.168.1.15: 11000000.10101000.00000001.00001111
```

```
192.168.1.1/32          covers 192.168.1.1
192.168.1.2/31          covers 192.168.1.2-192.168.1.3
192.168.1.4/30          covers 192.168.1.4-192.168.1.7
192.168.1.8/29          covers 192.168.1.8-192.168.1.15
```

Listing 2.3: IP addresses range 192.168.1.1 to 192.168.1.15 in CIDR notation.

```
0***1                   covers numbers 1,3,....,15
0**10                   covers numbers 2,6,10,14
0*100                   covers numbers 4,12
01000                   covers number 8
```

Listing 2.4: Range 1 to 15 in ternary representation with arbitrary wildcards.

If the classification system only supports ternary values, this range must be transformed first. One possibility is the common *Classless Inter-Domain Routing* (CIDR) block notation, which denotes IP networks by dividing an IP address in a network prefix (*most significant bits* (MSBs)) and a host identifier (*least significant bits* (LSBs)). The number of bits in the network prefix is given by a decimal number. For example, the CIDR block $192.0.2.0/24$ uses 24 bits for the network prefix and therefore contains all addresses from $192.0.2.0$ to $192.0.2.255$. An arbitrary range of IP addresses can be expanded to CIDR block addresses for ternary representation, as shown in Listing 2.3 for the above given example range.

In contrast to the longest-prefix match, where wildcards are only allowed to be filled without gap from the LSB, classification systems like TCAMs allow wildcards at arbitrary positions. In this case, the same range could also be represented as shown in Listing 2.4. In either case, the given range expands to four rules. This expansion is multiplicative if a single rule contains multiple fields with ranges. Therefore, a rule set containing rules with multiple range fields can grow for several orders of magnitude in size when implemented in ternary classification units like a TCAM [19, 93].

Software-defined Networking Hardware

Modeling and managing computer networks using the concept of SDN is a topic of major and ongoing interest [53]. SDN aims at providing an abstraction of network packet processing in several layers:

- The *application plane*, where the desired network behaviour is described using an abstract language,

- the *control plane*, where an SDN controller translates requirements from the application plane and is connected to the next lower layer,

- the *data plane*, where *network elements* represent the traffic processing elements. In practise, this logical entity is typically an SDN switching hardware.

Data plane devices like SDN switches usually rely on dedicated hardware packet classification engines (e.g., ASICs with TCAMs) to process packets and match them against flow tables and rule sets [64]. The advantage of using SDN hardware lies in the standardization of the communication protocol between the layers, which is—to some extent—independent of the actual type of hardware classification system [53, 61]. The increasingly popular language *P4* can be used to describe network behaviour on the data plane [16]. Moreover, dedicated hardware units with support for P4-compiled programs are available and promise fast implementation [116].

Other Hardware-based Approaches

Due to their highly parallel processing capabilities, *graphics processing units* (GPUs) have also been evaluated for network packet classification [91]. Finally, dedicated NPUs [57, 70] are essentially ASICs that implement optimized packet classification structures while also providing a certain level of programmability [48, 49, 57, 70].

The downside of hardware approaches is the limited functionality compared to software firewalls, e.g., only stateless matching support. It should be noted that such features can be implemented in hardware (like stateful IDSs [79]). However, their implementation requires significantly more effort when compared to software approaches [22, 84]. In contrast to software, hardware components cannot be dynamically loaded on demand. Therefore, a hardware design needs to directly implement all possibly needed functionalities. In this thesis, we use hardware-based packet classification systems of different kinds—the concepts are portable and can be realized with different approaches. In addition to the introduced generic variants, we also evaluate using further optimized, rule set specific hardware classification circuits.

2.3.4 FPGAs for Network Processing Applications

FPGAs are a certain type of re-programmable integrated circuit. They allow designers to directly describe the logical structure of the circuit without the need to undergo a full hardware design cycle as it is necessary for ASICs. This way, they can exploit advantages of dedicated hardware circuits while maintaining the flexibility to alter the circuit. An FPGA achieves this by its generic layout, where standard logic elements can be connected according to the circuit description. At the time of writing, FPGAs can provide several million logic gates [103]. These devices are usually programmed by describing the circuit using a language like *VHSIC Hardware Description Language* (VHDL) or *Verilog* [42, 43]. This description is then translated in several *synthesis* and *implementation* steps into a vendor- and device-specific configuration, which is often called the *bitfile*.

A typical FPGA comprises of several main elements [51]:

- *Logic Blocks* are the main logic resource and contain a standard set of logic cells, like *lookup tables* (LUTs) and *flip-flops* (FFs). These cells can be configured and connected to each other using a dedicated chip-wide routing infrastructure. The LUTs can also be used as dynamic memory elements for the design, this type of memory is called *distributed* random-access memory *(RAM)*. Depending on the vendor, logic blocks are often called *configurable logic blocks* (CLBs) or *logic array blocks* (LABs).

- *Hard Blocks* or *Hard IP Core*: As using generic logic blocks for building standard functions may be inefficient or not feasible, modern FPGAs also provide several hard IP cores (note that IP stands for *intellectual property* in this case). These blocks are essentially optimized hardware circuits which can be connected to and used together with the FPGA's logic elements. Typical functions that are implemented in such blocks are transceivers, due to the higher clock rate requirements. Processor cores like, e.g., an *ARM Cortex* can also be found on FPGAs. Although it would be possible to implement a processor architecture with logic blocks (called *soft core*), such hard IP cores achieve a much higher density on the chip. Another important type of hard blocks are internal memory blocks (e.g., RAM) or controllers for external memory. Both enable access to larger quantities of memory in comparison to distributed memory.

- *Clocking and Reset*: One of the main challenges in FPGA design is the timing. All signal propagations are subject to delays due to logic transitions and physical path delay. In the standard case of *synchronous design*, the config-

ured circuit must ensure that all signal propagations are completed before the corresponding edge of the clock signals triggers the next cycle. An FPGA typically provides one or more dedicated clock and reset networks, which will span a *clock domain* across the design or parts of it. The generation of clocks is also implemented in dedicated blocks, in order to provide stable and configurable sourcing of these networks.

Since the emergence of FPGAs in the mass market in the 1990s, they continued to gain in performance and found new target applications [101]. Today they are used, amongst others, for image processing, broadcast and radio communications, security, and hardware acceleration. FPGAs are also used for networking applications. For devices like switches and bridges, they provide the flexibility of the reconfigurability, but can still achieve the high throughput and low latency within the range of an ASIC. The advantages in performance and availability of standardized platforms made them also suitable for more advanced networking applications [20]. A popular example is high frequency trading on the stock market, where extremely low latency in network packet processing is required for competitive results [59]. This cannot be achieved with software based systems. Although ASICs would possibly allow even lower latencies, they are not a realistic option as the trading algorithms are adapted frequently and need to be implemented as quickly as possible.

Further examples demonstrate FPGAs being used for packet classification [31, 48, 49]. Most of these works focus solely on stateless packet classification, as dynamic state lookups are a challenging task on FPGAs [79]. FPGAs can also be used for IDSs [11, 22, 84]. An IDS typically utilizes static string matching techniques, which have also been widely discussed as FPGA implementations [84, 85].

The reconfigurability of an FPGA can be exploited to use the actual characteristics of the rule set for optimizations during synthesis and implementation, resulting in more efficient circuits that are also evaluated and used in this thesis [39]. Newer FPGAs and development tools further allow a feature called *partial reconfiguration*. Using this feature to dynamically replace an optimized packet classification system as part of an FPGA design during runtime has also been briefly evaluated, but showed no significant reduction of the update latency in comparison to regenerating the full FPGA configuration. A major part of this thesis is dedicated to hybrid combinations of different classification systems, rather than optimizing the approaches themselves.

A typical FPGA networking platform like it is used in this thesis is either a standalone board or a plug-in card like the peripheral component interconnect *(PCI) Express* (PCIe)-based NetFPGA [96]. Using the example of the NetFPGA SUME,

the data path begins with the transformation of incoming data from external interfaces and a consolidation to a single internal pipeline. This internal data pipeline operates on Ethernet frames. Within this pipeline, the desired operations and modifications can be carried out. Afterwards, the packet is passed on to an outgoing interface, which can either be an Ethernet interface or the host system via *direct memory access* (DMA).

2.3.5 Hybrid Packet Classification Systems

The diverse characteristics of the different types of packet classification systems have inspired hybrid networking systems that combine software systems with fast hardware classification systems. For packet filters, a combination of an FPGA with `netfilter` has been shown in [21]. The use of an FPGA as a hardware acceleration extension for a software-based IDS has been shown in [92]. A similar approach based on a dedicated NPU in conjunction with `netfilter` was described in [7]. Both use the hardware entity to offload connections acknowledged or set by the software. They do, however, not provide concepts how such a hybrid system can be used in order to make best use of each of the compartments capabilities, e.g., using the hardware for simple tasks and reverting to the software for complex decisions. Further, the software firewall is largely agnostic about the hardware classification process once connections were offloaded. Classification results of the hardware classification system are only available to the hardware system itself and are not forwarded.

The hybrid design in this thesis follows another approach: instead of classification and offloading of flows, the firewall rule set is analysed with regard to the characteristics of the rules. The rules are then distributed to the hybrid compartments so that the overall classification result corresponds to a standard, non-hybrid classification system using the full rule set. Several challenges arise for this type of distribution, some of which have been described for distributed firewall systems with regard to rule set anomalies in [80]. Special care must be taken for stateful packet filtering rules that refer to connection states of other rules. Software firewalls like `netfilter` use an implicit behaviour, i.e., stateful rules may also refer to a state that was established by an actually stateless rule. Such dependencies must not be disregarded if the semantic of the rule set should be identical. Moreover, instead of using only generic hardware classification systems, we evaluate our approach using logic-level optimized packet classification circuits on FPGAs. The usage of our own hardware design further enables forwarding of hardware classification results to the software classification system.

2.4 Summary

The development of networking devices is driven by the continuously increasing demand for higher throughput. For most of these devices, the packet classification problem is the defining challenge: in order to determine the action for each packet, a set of header fields is matched against a rule set. A variety of approaches have been introduced for this task—reaching from dedicated, specialized hardware to software-based systems running on standard hardware, all of which having their specific advantages and disadvantages. Table 2.1 provides a qualitative overview of some state-of-the-art examples that have been described in this Chapter.

Approach	COTS	Open Src.	Flexibility	Analysis Features	HW-Offl.	Perf.
Linux netfilter	Yes	Yes	High	Comprehensive	None	Limited
Linux eBPF	Yes	Yes	High	Comprehensive	None	Limited
DPDK/pfSense [63]	Yes	Yes	Limited	Limited	None	Up to line speed
The shunt [92]	Partly[2]	Partly[3]	Medium	Comprehensive	Flows	Up to line speed
NFShunt [62]	Yes	Yes	Medium	Comprehensive	Flows	Up to line speed
FPGA stand-alone	Partly	Partly[3]	Medium	Limited	Full	Up to line speed
Com. HW firewall	No[4]	No	Limited	Varies	Full	Up to line speed

Tab. 2.1: Qualitative comparison of different packet classification approaches for firewalling.

[2]Modifications required to standard software components.
[3]Design contains closed-source components.
[4]Vendor-specific, not interchangeable.

In the following chapters, we will use two of these approaches as a basis for the hybrid firewall HyPaFilter: `netfilter`, the standard software firewall on Linux, and an FPGA-based hardware classification system. The latter was chosen to allow the implementation of high-performance classification circuits, while offering the flexibility for specific extensions. One of the design goals was to further avoid changes to standard software components like the firewall itself and only using built-in features. The HyPaFilter approach will also demonstrate how hardware classification results can be used in the software classification process without the need for modifications to the software firewall itself.

3 HyPaFilter: A Hybrid Firewall

This chapter is based on previous work by the author [2, 5]. The strategies presented in Section 3.4 and their implementation originated from collaborative discussions and should also be attributed to fellow co-authors. The implementation of software user interface and management tool was accomplished by Sven Hager.

3.1 Overview

Most firewall systems that can be seen in practise are either software-based firewalls like `netfilter/iptables` [127], `pf` [119], or `ipfw` [111], or otherwise hardware-based systems from commercial manufacturers [113]. The advantages and challenges of the different types were already discussed in Chapter 1 and Chapter 2. Apart from raw performance, the level of security is a practical issue and of major interest for firewall systems. Although hardware-based systems like [113] are incorporating an increasing amount of features – often described as *next generation firewalls* (NGFWs) – formerly only available in software-based systems, the depth of analysis is not comparable to application layer gateways [107]. Working evasion techniques for popular NGFWs have been shown in various flavors [128]. Another issue is the fallback action that is executed once firewall systems approach their limits, e. g., for the maximum number of states. Technical documentation [106, 112, 120] suggests that *default open* is a commonly used method to avoid network outages due to overload. Not least, the high costs for hardware-based firewalls hinder a wide-spread use—costs that will further increase if more capabilities are integrated. Therefore, it would seem more rational to use hardware classification for simple classification tasks only, hereby keeping matching circuit complexity and implementation effort on a practical level.

In this chapter, we will answer the first research question (regarding a combination of two packet classification systems) by describing a newly introduced concept for a hybrid firewall that combines the advantages of massively parallel matching hardware and powerful inspection capabilities of software-based packet filters. The HyPaFilter approach aims to reach the packet rate and processing latency of a dedicated hardware firewall for common, easy to classify traffic, while providing the flexibility and functionality of a software firewall for packets which require complex processing. The hardware classification system is intended to be implementable on various types of FPGAs and is therefore constrained to a reasonable complexity and depth of analysis. HyPaFilter partitions a user-defined packet processing policy into two parts. First, a hardware part which only contains rules that can be handled by the specialized matching hardware. Secondly, a software part, which is handled by the software firewall and contains rules that

incorporate complex decisions or that are for other reasons required to determine the correct classification result. Incoming packets are always analysed by the hardware firewall first, so unambiguous decisions by the matching hardware can be carried out quickly. Further, the hardware part has the ability to redirect packets to the software firewall, if necessary. In the following, we will refer to this redirection as *shunting* packets (not to be confused with [92], where the software *intrusion prevention system* (IPS) shunts approved flows by putting them into the hardware's flow table). We found that a key challenge in such a hybrid design, regardless of its concrete implementation, is the proper handling of dependencies between different rules in the specified policy: if the hardware detects a rule match of an incoming packet in the hardware part of the policy, it must ensure that the packet does not match a more highly prioritized rule installed in the software filter *before* the action specified by the hardware-detected rule is applied. However, it is desirable to avoid a full-fledged software packet classification whenever possible in order to achieve the full hardware speedup for a large number of packets. In order to overcome this challenge, the HyPaFilter approach determines the largest rule index in the simple rule set up to which a hardware-only classification is safely possible.

Furthermore, even if complex processing for a packet is required, the matching information from the hardware can be reused in order to narrow down the set of rules the software filter has to match against this packet. Another difficulty that a hybrid classification system has to cope with are policy updates: in a real world setting, both the simple and complex parts of the rule set may change at any time.

In summary, we must solve the following key challenges for our hybrid firewall:

1. partitioning rule sets without changing the semantics of the original rule set,

2. avoiding the slow path (software) as much as possible,

3. avoiding redundant classification effort, and

4. handling updates to the rule set.

As FPGA-based systems are suitable candidates for high performance, low latency network applications [69], we prototyped the HyPaFilter approach using a dedicated FPGA networking platform, the NetFPGA SUME [96]. For the software firewall, we are using a standard Linux host with `iptables/netfilter`. In this setup, the NetFPGA SUME is initially configured with tailored logic that matches

packets against every simple rule in parallel, allowing it to perform basic firewalling tasks without involving the host at all at speeds of up to 40 Gbps for 64 byte frames. Complex rules and policy updates are implemented in `netfilter` in order to allow for comprehensive packet analysis as well as short rule update latencies. Whenever possible, updates that involve simple rules are moved from the software filter to the hardware filter during the next hardware configuration cycle.

The achievable performance of HyPaFilter depends on both the structure of the implemented policy as well as on network traffic characteristics. However, previous examination of real-world traffic in [7] showed that the fraction of traffic which can be analysed by simple packet filter rules is large enough to expect a significant performance gain in practical applications. Our evaluation results indicate that the HyPaFilter system can significantly increase the maximum achievable classification throughput over a software-only approach even for policies with many and widely scattered complex rules. In the current state of development, stateful firewalling relies on the software firewall only. Further, stateful firewalling requires an explicit notion of all state-tracking functionality in the rule set, i.e., all rules affecting or tracking states must be written as stateful rules by using the correspondent parameter. Handling implicit state-tracking in a manner that is identical to the behaviour of `netfilter` will be discussed in Chapter 4. The intended use case prefers scenarios like bridging firewalls, denial-of-service protection, or demilitarized zone configurations, where many policies can be implemented by stateless firewall rules.

In the following sections, we will describe in detail the hybrid setup and the strategies used for separating the different types of rules. Further, we will evaluate how the policies affect the throughput.

3.2 Related Work and Research Gap

Two important categories of research efforts are *classification algorithms* and *hardware architecture*, both of which have been described in Chapter 2. HyPaFilter employs *rule set transformation* techniques, which are orthogonal to the employed classification algorithm/architecture. The goal is to transform an initial rule set $\mathcal{R}$ into an equivalent rule set $\mathcal{R}'$ which can be traversed faster for incoming network packets. Existing approaches for rule set transformation are common, e.g., rule set minimization [56]. Another approach is the encoding of decision tree data structures into the rule set [38], which reduces the number of rules that must be traversed for the classification in comparison to a plain linear search approach.

HyPaFilter utilizes a variant of a decision tree optimization to install rules in the software filter. In contrast to directly using the rule's classification fields for building the tree, hardware matching results are exploited. As all packets are classified by the hardware classification system first, a preliminary classification result is available for each packet. This result can be reused in the software firewall to search through the tree branches, each containing pre-calculated groups of rules.

The packet filters shown in [7, 21] demonstrate the feasibility of hybrid approaches, but do not answer the following key questions:

1. How should a packet processing policy be deployed in a hybrid system in order to reach high classification performance while avoiding changing the semantics?

2. How does the hybrid system implement rule set updates?

In order to provide an answer to these questions, we present three rule set partitioning schemes as well as update mechanisms to handle rule set changes.

3.3 Hardware Classification Compartment

In order to support good classification performance, short rule set update latencies, and expressive rule set semantics, the HyPaFilter system relies on a hybrid matching algorithm that first processes every incoming packet on the FPGA. After the packet is matched, the FPGA circuitry decides whether the packet requires further complex processing in the host-based `netfilter` system.

The classification system implemented on the FPGA solves the packet classification problem on the simple rule set $\mathcal{R}_S$, as introduced in Section 2.1 and Section 2.2. It therefore implements every simple rule $R_i \in \mathcal{R}_S$. In order to achieve high matching performance on the FPGA with a low, deterministic processing latency per packet, we decided to use a *massively parallel filtering circuit* (MPFC). An MPFC is a rule-set-specific parallel matching engine, which is generated by translating every simple rule $R_i \in \mathcal{R}_S$ at setup time into a specialized match unit $M_{\mathcal{R}_S(i)}$ specified in the VHDL, building upon the technique proposed in [39]. Recall that $\mathcal{R}_S(i)$ is the index of rule R_i within the sublist $\mathcal{R}_S$. This process is illustrated in Figure 3.1. Since each rule in $\mathcal{R}_S$ is a conjunction of simple checks, such as subnet tests or port range tests, the match units are composed of a small number of basic comparator circuits. For example, a rule which matches TCP packets if the source

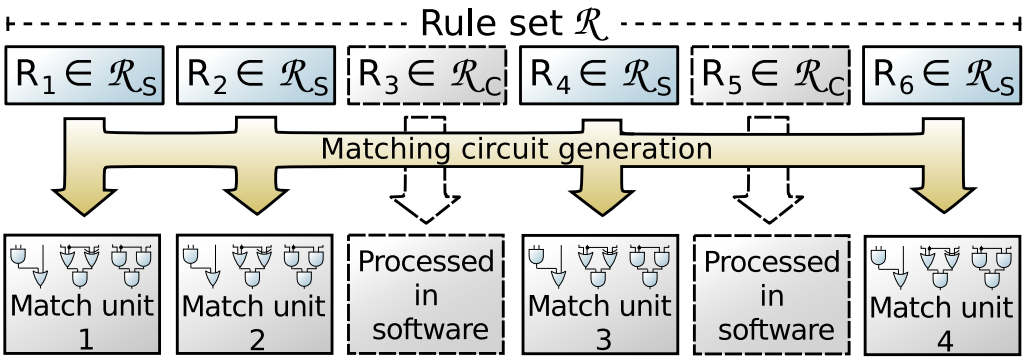


Fig. 3.1: Translating simple rules of Listing 2.2 into match units.

IP address is in the subnet $203.0.0.0/8$ with destination port 80 is translated into three specific comparator circuits: the first one compares the packet's transport protocol field against the TCP transport protocol number 6, while the second and third comparators compare the first octet of the packet's source IP address against 203 and the packet's destination port against 80, respectively. Finally, the results of these comparators are ANDed to determine whether the rule matches.

As the match units are arranged in parallel, incoming network packets can be matched against the entire simple rule set $\mathcal{R}_S$ in a single clock cycle, which yields a result bit vector V_{res} of size $|\mathcal{R}_S|$. Here, the entry at position $\mathcal{R}_S(i)$ of the result vector V_{res}, which we denote by $V_{\text{res}}[\mathcal{R}_S(i)]$, stores a 1 if rule R_i matches the current packet, and a 0 otherwise. As we are interested in the most highly prioritized matching rule, we employ a priority encoder to determine the index of the first enabled bit in V_{res}, which we will refer to as *match_index*. The hardware matching process is sketched in Figure 3.2.

We opted to use the above described *logic-level optimized* (LLO) MPFC in our FPGA-based prototype system due to its small hardware resource footprint. In comparison to generic hardware matching techniques with comparable throughput, such as StrideBV [33] or TCAMs [73], tailor-made matching circuitry is significantly smaller and dissipates less power when implemented on an FPGA [1, 37]. During our experiments, the design toolchain was not able to generate a native TCAM implementation on the FPGA for capacities as low as 100 IPv6-capable rules without timing errors, while the tailor-made matcher could support several thousands of rules. Nevertheless, we point out that it is of course possible to step away from an FPGA implementation platform and instead use a different hardware matcher for simple rules, e. g., on the basis of a high-density ASIC-based TCAM.

Up to this point, the packet classification problem is solved for the simple rule set $\mathcal{R}_S$ solely in hardware, as the *match_index* can be used in order to quickly

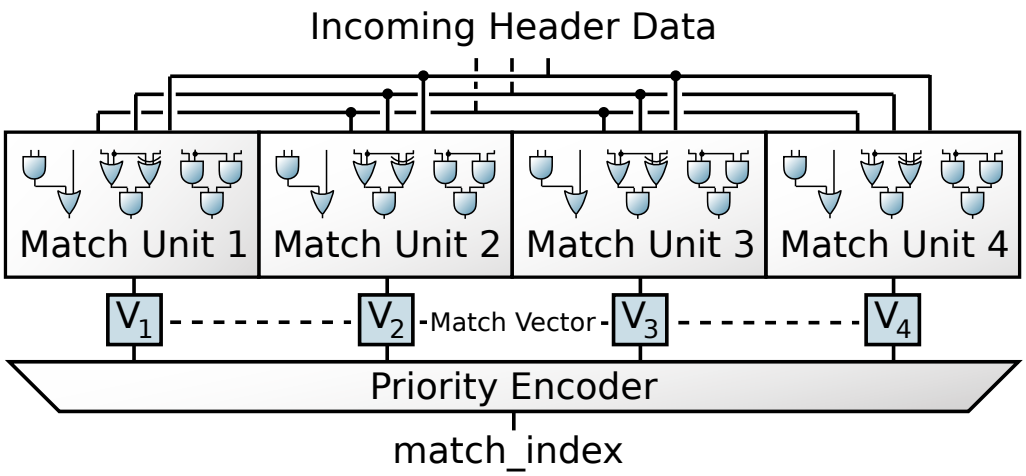


Fig. 3.2: Parallel match of packet header data against $\mathcal{R}_S$.

look up the action $A_{\mathcal{R}_S,P}$ that must be applied for the current packet P. If the installed rule set $\mathcal{R}$ does not specify any rules with complex checks, i.e., if $\mathcal{R}_C = \emptyset$ and thus $\mathcal{R}_S = \mathcal{R}$, then the classification is complete at this point and $A_{\mathcal{R}_S,P}$ is applied to the current packet. However, if $\mathcal{R}_C \neq \emptyset$, then additional processing may be required on the host system. Accordingly, some packets must be *shunted* from the FPGA device to the software filter to compute the correct classification result. However, as software-based classification of shunted packets is expensive (in comparison to FPGA-only packet processing), the number of shunted packets should be as small as possible. Furthermore, the shunting decision itself should be computed at line speed in order to not bottleneck the packet pipeline. HyPaFilter uses a simple shunting strategy that we call *index-based shunting*.

This technique, which was introduced in [2], partitions the simple rule set $\mathcal{R}_S$ into an *unambiguous* rule set prefix and an *ambiguous* rule set suffix. When the FPGA detects that an incoming packet P matches a rule in the unambiguous part, it is processed entirely in hardware. Otherwise, P is shunted to the host system for further processing. Here, a simple rule $R_i \in \mathcal{R}$ is called *ambiguous* if there exists a complex rule $R_j \in \mathcal{R}$ with $j < i$, otherwise R_i is called an *unambiguous* rule. Thus, a packet P is shunted to the software classification system whenever a complex rule in $\mathcal{R}_C$ installed on the host system *could* match the current packet P with a higher rule priority than the matching rule in $\mathcal{R}_S$. In the following, we will denote the smallest index of a rule in $\mathcal{R}_S$ with complex checks by the term *shunt_index*. We point out that this simple shunting strategy is agnostic to the actual rules itself, which means the strategy only uses the index (i.e., the priority) of the rule to determine a possible conflict. It is not further examined whether the rules in question actually interfere with each other, which can consequently result in a larger number of shunted packets than necessary.

In the example shown in Figure 3.1, the unambiguous prefix consists of the rules R_1 and R_2. In contrast, the ambiguous suffix contains the rules R_4 and R_6, as the complex rule R_3 is more highly prioritized and could potentially conflict with R_4 or R_6, respectively. For instance, consider the case that the hardware matching circuit for the rule set sketched in Figure 3.1 computes that *match_index* is 3 for an incoming packet P (that is, the packet matches the simple rule R_4). In this case, our hardware classification might be incorrect, as the complex rule R_3 could also match on the packet P. Thus, whenever *match_index* $\geq$ *shunt_index*, index-based shunting sends the classified packet to the host for further processing.

This technique computes the correct classification result in every case, since packets that might match a complex rule are always shunted to the host system. Moreover, changes in the complex part of the rule set only require an update of the register on the FPGA that holds the *shunt_index*. The hybrid operation can also bridge the delay that may occur when the hardware filter core needs to be updated. For this purpose, packets matching rules affected by the update are shunted until the change in the hardware becomes active. This allows the use of hardware filters where updates are costly in terms of time. Although this strategy is easy to implement, a major drawback comes into effect if complex rules appear with a high priority, which forces the shunting of all packets that match lower prioritized rules. In the worst case, a single complex rule at the top of the rule set would force all traffic to the host, which will consequently result in a setup equal to a standard software firewall. An improved shunting strategy addressing this constraint by analysing the rules for actual conflicts—in contrast to only using their priority—will be introduced in Chapter 4.

3.4 Software Classification Strategies

The task of the software filter running on the host computer (`netfilter` in our example) is to classify every shunted packet which cannot be handled exclusively in hardware. However, simply installing only the complex rule set $\mathcal{R}_C$ in the software filter is not sufficient, since shunted packets P could still also match simple rules in $\mathcal{R}_S$. This is the case when P is not matched by any complex rule with a higher priority than the first matching simple rule. As a consequence, the software filter must be able to reproduce the hardware classification result if the most highly prioritized matching rule is in $\mathcal{R}_S$ and not in $\mathcal{R}_C$.

The decision function f_{dec} is used to decide whether a packet needs to be shunted for this section. With index-based shunting, this function is a simple comparison

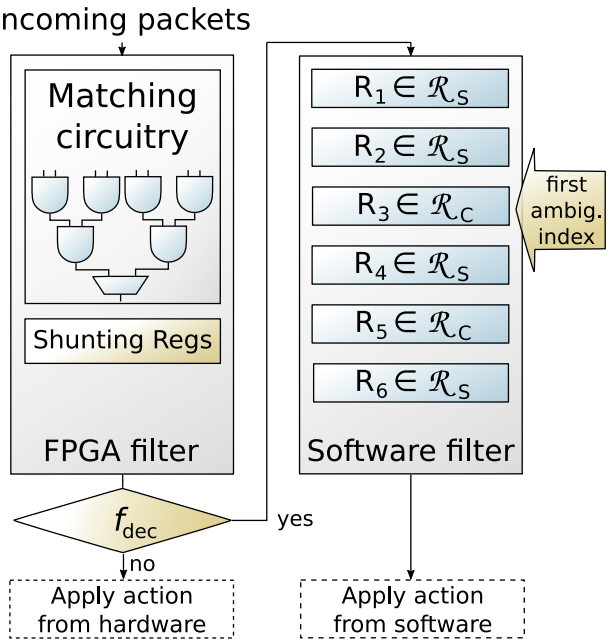


Fig. 3.3: Different strategies to implement the complex rule set $\mathcal{R}_A$ in the software filter: full set strategy.

of *match_index* $\geq$ *shunt_index*. In this section, we present three different strategies how the rule set in the software filter can be organized to achieve this goal.

3.4.1 Full Set Strategy

The most straightforward way to setup the software filter, which we call the *full set strategy*, is to simply install the entire rule set $\mathcal{R}$ in the software filter. That way, forwarded packets will always traverse rules in the correct order until the first matching rule is found, as sketched in Figure 3.3 for the example rule set from Figure 3.1. This approach allows for quick rule updates, since only one rule in the rule set installed in the software filter has to be changed in addition to a possible update of the shunting policies on the FPGA. This strategy is simple, but has a major disadvantage: the software filter may process a large number of rules for every shunted packet, including simple rules. It thus repeats significant work already done in hardware. This can be particularly expensive as, in contrast with the full-parallel match in the hardware filter, the rules are commonly processed linearly in software packet filters.

3.4.2 Cut Set Strategy

The amount of redundant work that is done in software for shunted packets can be reduced with a slight modification. Let β be an index such that a shunted packet

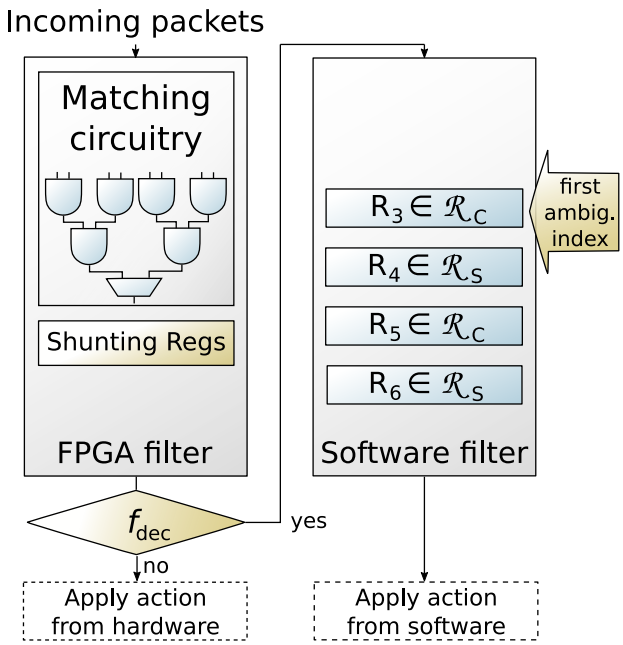


Fig. 3.4: Different strategies to implement the complex rule set $\mathcal{R}_A$ in the software filter: cut set strategy.

can never match a rule $R_i \in \mathcal{R}_S$ with $i < \beta$. In the case of index-based shunting, β is equal to *shunt_index*. For example, consider the rule set from Figure 3.1 and a packet P with *match_index* $= 3$. As *match_index* is equal to *shunt_index* , P will be forwarded to the software filter, which will superfluously once again test rules R_1 and R_2 against P. In order to avoid this potential extra work on the host system, the *cut set strategy* installs only rules in $\mathcal{R}_C$ and in $\{R_j | R_j \in \mathcal{R}_S \wedge j \geq \beta\}$ in the software filter, as sketched in Figure 3.4.

In comparison to the full set strategy, the cut set strategy has a higher rule update cost, as a potentially larger number of rules must be inserted or removed from the software filter in case of an update. However, evaluation shows that the update effort clearly pays off in terms of classification performance, as fewer rules must be traversed by shunted packets.

3.4.3 Interval Strategy

The strategies described so far implement rule sets in the software filter that are agnostic to the partial classification result tuple <*match_index*, $A_{\mathcal{R}_S,P}$> previously computed on the FPGA for every shunted packet P. This results in wasted effort on the software side and inflates the software-side rule set—also in case of the cut set strategy. An example can be seen in Figure 3.5. Here, the rule set contains two complex rules $\{R_3, R_5\} \in \mathcal{R}_C$. A packet that would match rule R_9 would therefore be shunted to the host and be matched again by the software firewall. In the

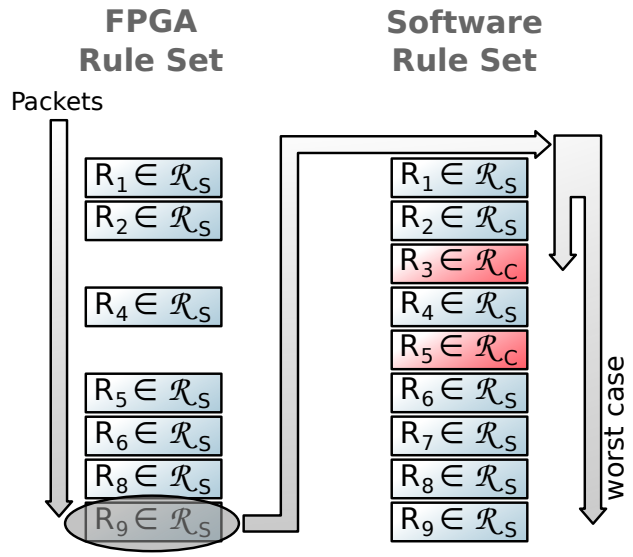


Fig. 3.5: Redundant classification effort for simple rules.

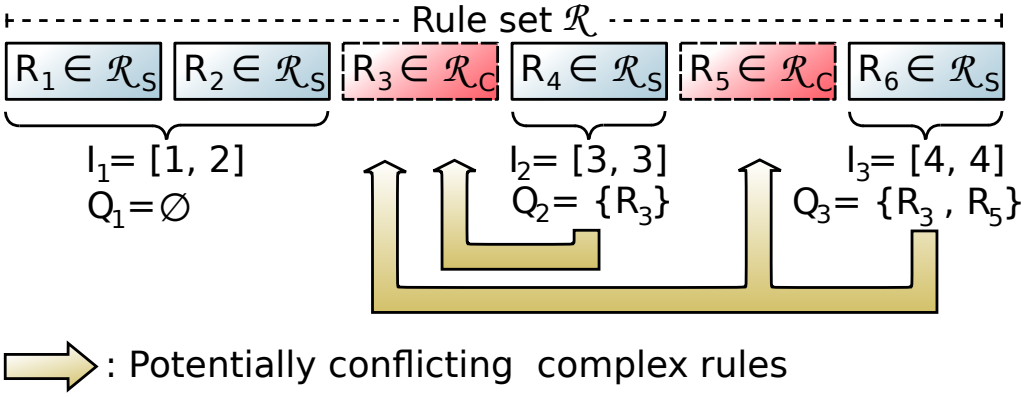


Fig. 3.6: Intervals in the rule set $\mathcal{R}$.

best case, this match would require a traversal of $\{R_1, R_2, R_3\}$ and $\{R_1, ..., R_9\}$ in the worst case, both for the full set strategy. This is partly redundant, since all simple rules, in this case $\{R_1, R_2, R_4, R_6, R_7, R_8, R_9\}$, have already been checked in the hardware classification unit. To avoid the re-computation effort, the *interval strategy* relies on metadata handed over from the FPGA to the matching software when a packet is shunted, i.e., the match index and action tuple <*match_index*, $A_{\mathcal{R}_S,P}$>. Simply put, the goal of the interval strategy is that shunted packets should only be tested against a subset of the complex rules $\mathcal{R}_C$ and none of the rules in $\mathcal{R}_S$ in software again.

The idea behind the interval strategy is that groups of consecutive simple rules $G_k = \{R_i, \ldots, R_{i+\alpha}\}$ in $\mathcal{R}$ can be mapped to intervals $I_k = [\mathcal{R}_S(i), \mathcal{R}_S(i+\alpha)]$, with $[a, b] = \{x \in \mathbb{N} \mid a \leq x \leq b\}$. For instance, the simple rules from the example rule set in Figure 3.6 form three groups $G_1 = \{R_1, R_2\}, G_2 = \{R_4\}$, and $G_3 = \{R_6\}$, with the corresponding intervals $I_1 = [1, 2], I_2 = [3, 3]$, and $I_3 = [4, 4]$. Each interval represents a range of match indices, which may be computed by

the FPGA for an incoming packet P. If P is shunted to the host, the *match_index* computed on the FPGA falls into exactly one of these intervals. The interval strategy exploits this fact by pre-computing the chain of complex rules Q_k for every interval I_k that could contain a more highly prioritized matching rule for a packet P whose hardware-computed *match_index* falls into interval I_k (i.e., P matches a simple rule in group G_k). In the example shown in Figure 3.6, Q_1 is empty, since there are no complex rules in $\mathcal{R}$ that are more highly prioritized than the simple rules R_1 and R_2. In contrast, $Q_2 = \{R_3\}$, as the complex rule R_3 is more highly prioritized than the simple rule R_4 and thus could match on packets that have been assigned to R_4 by the FPGA. Similarly, Q_3 is set to $\{R_3, R_5\}$, as R_3 and R_5 are more highly prioritized than the simple rule R_6. The function BUILD_RULE_SET_INTERVALS in Algorithm 1 describes the general construction of intervals and complex groups.

Algorithm 1 Build intervals and complex groups for interval strategy.

```
1:  function BUILD_RULE_SET_INVERVALS(Rule set R)
2:      I ← [ ] // set of intervals I
3:      Q ← [ ] // set of complex rule groups Q
4:      group_index ← 1
5:      in_simple ← false
6:      for Rule R_i ∈ R do
7:          if IS_SIMPLE(R_i) then
8:              // find the first simple rule of a consecutive group of simple rules
9:              if not in_simple then
10:                 // set interval start to the index of the rule in R_S
11:                 I_group_index,first ← R_S(i)
12:                 in_simple ← true
13:                 Q ← Q + Q_group_index-1
14:             if i = |R| then // stop if end of rule set is reached
15:                 I_group_index,last ← R_S(i)
16:                 I ← I + [I_group_index]
17:         else // R_i is a complex rule
18:             if in_simple then
19:                 // R_i is the first complex rule after a simple rule
20:                 in_simple ← false
21:                 // preceding simple rule is the end of the interval
22:                 I_group_index,last ← R_S(i − 1)
23:                 I ← I + [I_group_index]
24:                 group_index ← group_index + 1
25:                 Q_group_index ← Q_group_index-1
26:             Q_group_index ← Q_group_index + [R_i]
27:             if i = |R| then
28:                 Q ← Q + Q_group_index
29:     return (I, Q)
```

Algorithm 2 Look-up and build complex sub-group for packet P.

```
1: function GET_COMPLEX_SUBSET(I, Q, match_index, A_{R_S,P})
2:     for Interval I_i ∈ I do
3:         if match_index ≥ I_{i,first} & match_index ≤ I_{i,last} then
4:             set action of match-anything-rule R_* to A_{R_S,P}
5:             Q_P ← Q_i + R_*
6:             return (Q_P)
```

Whenever a packet P is shunted to the host, the FPGA driver fetches the tuple <*match_index*, $A_{\mathcal{R}_S,P}$> from the hardware. Then, the FPGA driver code on the host determines the index k of the interval I_k that contains the *match_index*. Without referring to the technical implementation of the look-up procedure, function GET_COMPLEX_SUBSET in Algorithm 2 formally describes how to resolve the complex sub-group including the default action for each packet. In HyPaFilter, before the actual `netfilter` packet classification starts, the index k, as well as the hardware action code $A_{\mathcal{R}_S,P}$ are written to the most significant 28 and least significant 4 bits of the `netfilter` `mark` field, which is a 32 bit metadata field attached to the packet P. With `netfilter` supporting tests on the `mark` field, we can use this information to achieve two goals: first, we want to limit the set of complex rules that must be tested in `netfilter` to only those that are more highly prioritized than the first matching simple rule. Second, we want to apply the hardware-computed action $A_{\mathcal{R}_S,P}$ in `netfilter` *without* the need to re-traverse any simple rule in software if there is no match in $\mathcal{R}_C$.

To this end, the rules that are installed in `netfilter` for the interval strategy are implemented as search tree, following an approach introduced in [38]. In HyPaFilter, this search tree is generated as follows: the `netfilter` rule set starts with a sequence of rules which implement a binary search over the interval index k encoded in the most significant 28 bits of the `mark` field, while the groups of complex rules Q_k are placed into separate chains. This allows `netfilter` to quickly locate the chain of relevant complex rules Q_k during the matching process, as sketched in Figure 3.7. Finally, each chain containing Q_k ends with fallback rules, which can either be a group of one rule for each possible action, or a jump instruction to an equivalent chain containing those rules. This fallback group uses the least significant four bits of the `mark` field to determine and apply the action $A_{\mathcal{R}_S,P}$ to the shunted packet P if no complex rule matches.

In comparison to the full set and cut set strategies, the interval strategy requires more complex preprocessing in case of a rule update, as the intervals for the complex rules have to be re-computed and communicated to the hardware driver. Furthermore, the `netfilter` binary search tree encoded in the filter rules must be re-generated. However, this strategy shows the best average classification perfor-

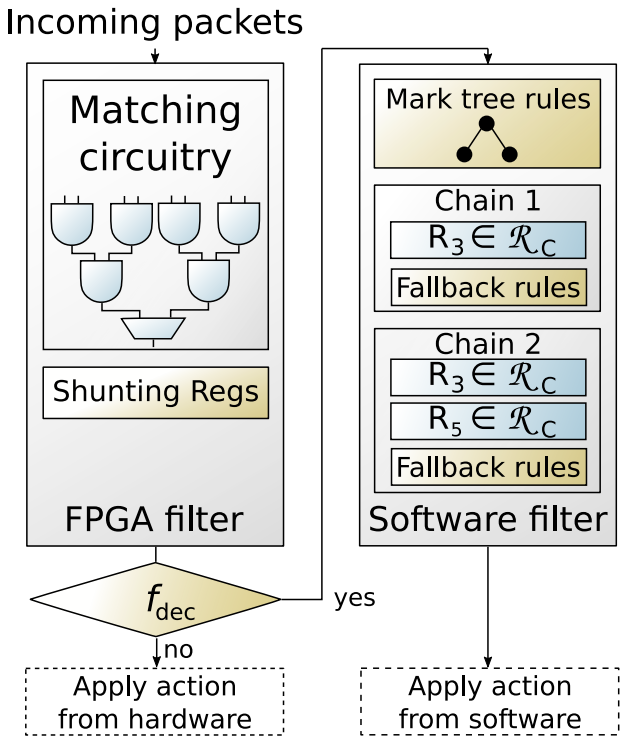


Fig. 3.7: Different strategies to implement the complex rule set $\mathcal{R}_A$ in the software filter: interval strategy.

mance in software when tested with our evaluation rule sets. Here, the number of traversed rules for each shunted packet P is orders of magnitude smaller than in the full set and cut set strategies. We point out that for some constellations, i.e., software rule sets containing only a few rules, the base structure of the interval strategy may also increase this number due to the inherent rules used for the tree structure. Nevertheless, the difference in these cases is negligible. Furthermore, this approach does not require a change of the `netfilter` source code in order to use the hardware-computed matching information. Instead, we completely rely on existing `netfilter` match functionality to accelerate the software matching process.

3.5 System Architecture and Operation

The prototype system for the HyPaFilter system consists of two functional units. One part is a standard host system, used to run the software firewall and the toolchain for managing the system. This can even be an already existing firewall appliance which needs to be upgraded in terms of performance. This system is extended by the second part, a general purpose FPGA addon card, as shown in Figure 3.8. These units must provide a sufficient communication path for

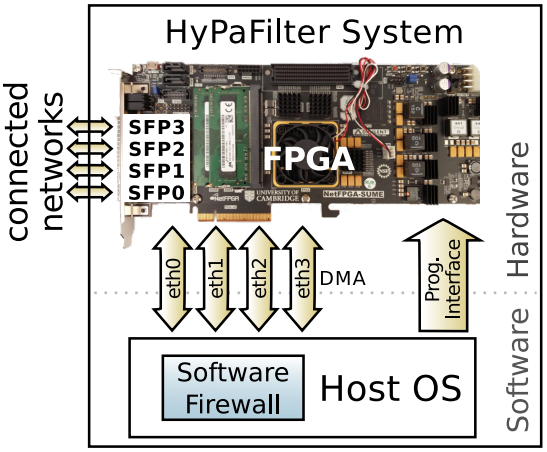


Fig. 3.8: Proposed structure of a HyPaFilter system. The host can be any COTS system capable of carrying the FPGA NIC.

transferring data and settings between them. The utilized plug-in card is a suitable FPGA platform that can provide the required interfaces to both communicate with external Ethernet networks as well as acting as a regular network interface card in regard to the host system. For our setup, we used the NetFPGA SUME [96]. It provides multiple network ports and can be plugged into a COTS system via PCIe. The card acts as the primary network interface connected to both the internal (e.g., LAN) and the external network (e.g., Internet). The hardware-based filtering is handled on the FPGA.

The host system carries the FPGA NIC and communicates with it via PCIe. The host runs the operating system where the software firewall `netfilter` with `iptables` is installed. It also supplies the tools to configure the FPGA, and provides a user interface for administrating HyPaFilter.

The host and the FPGA card are connected through several communication channels. For quick and simple configuration settings, the host system is able to set and read predefined 32 bit registers on the FPGA via PCIe. Network traffic between FPGA and host is handled via DMA. On the host side, a driver provides the functionality and interfaces so that the operating system can access the FPGA like a regular NIC. This is important since we do not want to rely on non-standard customizations to `netfilter` for HyPaFilter to work. By using a programming interface, the configuration of the FPGA can be updated. We used the Xilinx Vivado software toolchain [129] to generate the FPGA configuration based on a given rule set.

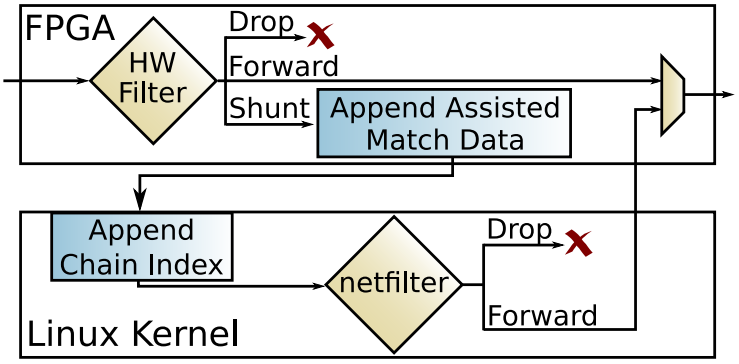


Fig. 3.9: Flow of packets through the HyPaFilter system.

System Operation

Incoming packets received from any connected network are first matched against the rules implemented on the FPGA. Based on the result and its validity with respect to the rule dependencies, packets are either dropped, forwarded directly (without interaction of the host system), or shunted to the host for further processing. Whenever a packet is shunted, the matching information—*match_index* and hardware action—is added to the packet in a driver-readable metadata field, which is needed for the interval strategy.

For outgoing packets from the host system, the FPGA NIC acts like a standard NIC. Such packets, e.g., packets shunted and processed by the software firewall or packets generated by the host itself, are therefore sent out through the corresponding network interface without further analysis. The packet flow is sketched in Figure 3.9.

Note that in some rare cases, shunting can lead to packet re-ordering, if the rule set is configured to shunt only a part of the flow's packets, e.g., distinguished by additional header fields. Re-ordering is allowed for IP packets, but should be avoided due to, e.g., negative effects on the performance of the TCP [13]. However, rule constructions where this can occur are rather theoretical. Flows that are targeted with rule sets that only test for fields examinable by the hardware matcher are either always shunted or processed in hardware. In fact, in all real rule sets we analysed during our evaluation (see Section 3.6.1), no packet re-ordering takes place.

The administrator needs to be able to manage the system without the need to understand the underlying complexity. In our implementation, we created a Python command line interface management tool. The general workflow for using HyPaFilter is shown in Figure 3.10.

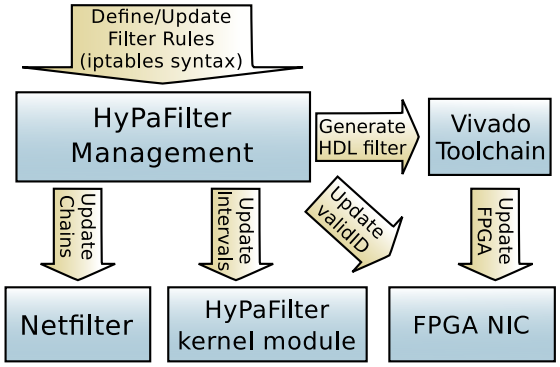


Fig. 3.10: HyPaFilter workflow with the management tool.

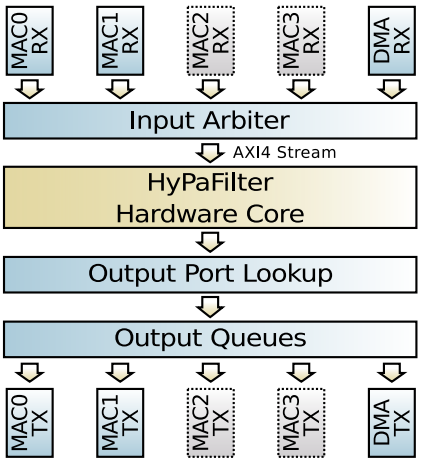


Fig. 3.11: Simplified dataflow structure of the NetFPGA SUME. The dashed elements are available, but not used in our evaluation.

Packet Data Path

The data flow through the FPGA can be shown in two layers. The underlying structure for general networking and communication tasks is based on the NetFPGA SUME pipeline [58]. The actual core that is responsible for filtering is embedded into this pipeline and connected via the AXI4 stream protocol [97] as shown in Figure 3.11. Internally, the HyPaFilter core uses a data bus width of 512 bits and runs at 180 MHz. Hence, the theoretically achievable throughput of 92.16 Gbit/s is enough to fully saturate all four 10 Gbit/s Ethernet ports including protocol overhead. The NetFPGA SUME currently uses a bus width of 256 bits which is converted before and after the hardware core.

Packets coming into the hardware core are first distributed (cloned) into a classification path and a data path, with the latter being a simple FIFO queue of 64 kB. In the classification path, the *Header Parser* extracts relevant information from incoming packets. For a versatile operation, the header parser must take care of the data alignment due to VLAN tag-stacks or various variable-length headers.

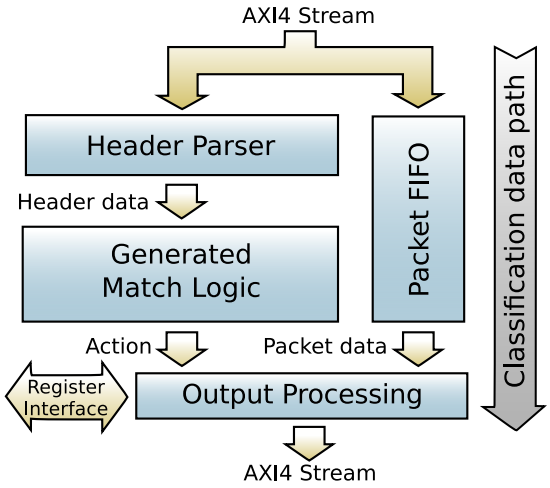


Fig. 3.12: Dataflow inside the HyPaFilter hardware core.

Therefore, it is implemented as a multi-stage non-blocking pipeline architecture. The preprocessed data is forwarded to the filtering module, which is generated by the management toolchain. After the classification, the decision is forwarded to the *Output Processing*, where the determined action is executed: DROP (read from FIFO and discard), FORWARD, or SHUNT by adapting the output port field in the packet's metadata. The register interface can be accessed from the host directly via PCIe. Figure 3.12 shows the described parts in the module.

As described in Section 3.3, the matching logic is able to classify packets in constant time. Since the hardware filtering logic contains no components that could cause a data-pipeline stall, it is clear that the HyPaFilter hardware core is never the limiting factor for raw data throughput in this setup. Hence, a reader might note that the separation into data and classification path yields no advantages in terms of maximum throughput. However, as we aim to support more complex decisions with possibly non-deterministic lookup time in hardware, this structure allows for more flexible development. The hardware filter core is able to extract and classify incoming packets against a variety of parameters like IP addresses, protocol fields, MAC addresses, port fields and several flags.

3.6 Evaluation

In the evaluation of HyPaFilter, we focus on four important performance metrics for packet classification architectures: packet rate, network latency, rule set update latency, and consumption of resources. This stands in contrast to raw data throughput measurements, which are more targeted at the data flow structure. The following experiments were conducted:

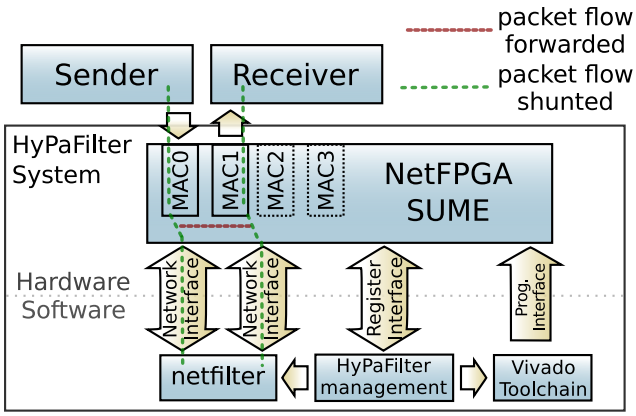


Fig. 3.13: Evaluation setup showing the relevant components. Traffic is generated on the sender and directed through the bridging HyPaFilter firewall.

- determining the maximum number of rules which can be implemented into the FPGA,
- measuring the maximum packet rate of the NetFPGA SUME architecture,
- measuring the performance of the HyPaFilter system and comparing the impact of rule updates using different strategies,
- measuring the network latency,
- measuring delays of the update process and number of rules and
- comparing against a commercial OpenFlow SDN firewall setup.

To generate a high workload on the classification engine while avoiding influence of handling large payloads via PCIe, we used small packets at a high rate. Packets carry just five arbitrarily chosen bytes as the application layer payload. For our evaluation, we set up a typical bridging firewall scenario as shown in Figure 3.13.

Traffic is generated and received by two dedicated machines, whose details can be found in Table 3.1. We generated rule sets and traffic using the ClassBench suite [89], which is widely used in this context. The system is easily capable of saturating the connected networks with traffic. These sender and receiver hosts are connected to the HyPaFilter system via optical fibre. We counted the number of packets received by the MAC-Core `MAC0` on the NetFPGA and those arriving on the network interface of the receiver. Further network connections between the systems to remotely start the test cycles and collect the results are not shown.

Sender/Receiver Configuration	
Host OS	Ubuntu Linux
NIC	Intel 82599ES Dual 10 Gbit/s
CPU	Intel Core i7-7700K
FPGA Host Configuration	
Host OS	Ubuntu Linux
NIC	Intel 82599ES Dual 10 Gbit/s
CPU	Intel Xeon E5-1650 v3
FPGA PCIe Card	NetFPGA SUME release 1.0.0
Development Software	Xilinx Vivado 2014.4
Software Firewall	`netfilter` and `iptables` v1.4.21

Tab. 3.1: System configuration.

The HyPaFilter host system carries the FPGA plug-in card and serves as the back-end software firewall. Its details can also be found in Table 3.1. The hardware filter core is integrated into a modified data pipeline based on the reference NIC project of the NetFPGA SUME.

3.6.1 Test Rule Sets

To evaluate the classification performance under replicable conditions, we generated our test rule sets with ClassBench [89]. The number of rules we could fit onto the FPGA was limited by the timing constraints and resulted in a maximum of 1100 rules. For evaluating our classification algorithm and strategy, we created three different UDP rule sets `acl1k1`, `fw1k1` and `ipc1k1`, with all rules applying the action ACCEPT. This way, the number of dropped packets can be regarded as the packet loss solely due to the architecture. As we only evaluate stateless classification performance, no state handling is performed and the network protocol of a packet has no influence on the processing speed.

ClassBench's *trace_generator* was used to generate trace files corresponding to the rule sets, using the parameter sets provided in [100]. A C program was developed and used for generating and transmitting the test packet stream from such a trace file. Besides, we were granted access to three confidential real-world rule sets by one customer of the computer security company *genua*.

For each test, the rule set was processed by the HyPaFilter management tool, integrated in the NetFPGA SUME pipeline, and afterwards synthesized and implemented into an FPGA configuration bitfile. Table 3.2 shows the resource utilization of the FPGA configuration for a Virtex 7 690T. The relevant parameters are usage of FFs, LUTs, LUTs used as memory elements (Memory LUT), and *block*

Resource	acl1k	fw1k	ipc1k
FF	9.12% / 0.86%	9.12% / 0.86%	8.26% / 0.86%
LUT	15.07% / 1.69%	16.22% / 2.85%	13.38% / 1.83%
Memory LUT	1.07% / 0.01%	1.10% / 0.01%	1.10% / 0.01%
BRAM	16.73% / 2.72%	16.73% / 2.72%	14.01% / 2.72%

Tab. 3.2: FPGA resource utilization overall/HyPaFilter core with different rule sets. Variance between different runs is negligible.

random-access memory (BRAM). Differences can be caused by the different rule sets and heuristic algorithms used during the implementation process in Vivado.

To measure the impact of changes or occurrences of complex rules to the rule set, we added in each test new rules to certain positions, starting from the end of the rule set. We used the following policy:

```
-m string --algo bm --string BAD -m statistic\
 --mode random --probability 0.99
```

This rule makes use of several modules that are available in `netfilter`, i.e., string matching and statistical analysis. However, the implemented policy itself is a toy example, as its main purpose is to implement a demanding rule in the rule set which cannot be handled by the hardware classification system. To evaluate the effect of the properties of these complex rules, we repeated the performance tests with "best-case complex rules" that had no complex operation (i.e., they artificially enforced shunting without additional processing steps in software). We found a negligible average performance increase of 0.036 pp ($\sigma = 0.15$). Therefore, the nature of the complex rules is not important for our evaluation.

3.6.2 Impact of Packet Shunting

In our first experiment, we measured the impact of shunting packets to the software. As processing packets directly in hardware without any software interaction provides the lowest latency and highest packet rate, we expect to achieve the best packet rate if no packets are shunted at all. If the fraction of packets that are shunted to the host increases, a drop in the obtained packet rate is to be expected. For this test, the FPGA NIC was configured to forward a certain number of the packets directly to the target, while the other packets were shunted to the host. To exclude additional processing overhead on the host system, all shunted packets were directly software-bridged to the outgoing interface without software firewall interaction.

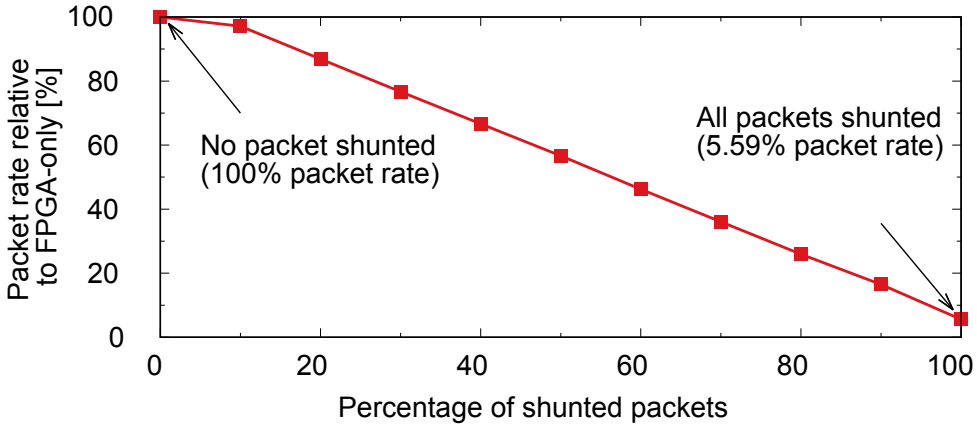


Fig. 3.14: Impact of shunting packets without firewalling, standard deviation too small to be visible.

We compared the number of ingress packets on the FPGA to the number of packets received at the receiver. Each data point shows the average packet rate of ten test runs, each lasting for 20 seconds. The standard deviation is too small to be visible. Figure 3.14 shows the results of this experiment. When all packets are forwarded by the FPGA, the test setup is capable of operating at line speed. With more packets being shunted to the software, the performance drops continuously. When all packets are shunted, the NetFPGA SUME basically acts as a simple NIC with a simple DMA engine, no offloading mechanisms are utilized. In this case, only 5.59% of the packets can be handled by the evaluated system. This demonstrates that in a hybrid system like the proposed one, FPGA-to-host communication is expensive and should be avoided in order to reach line speed performance. In comparison, the detection and output queue assignment of a shunted packet extends the processing pipeline by just one clock cycle, which has no measurable effect on the throughput or packet rate.

3.6.3 Architecture Packet Rate

Forwarding packets directly in hardware provides the lowest latency and highest packet rate. Therefore, the first experiment was used to measure the maximum packet rate dependent on the percentage of packets shunted to the software. We will later compare the packet rate of the different strategies against these values.

As the generated packets by the sender will match the rule set at certain positions with a predefined distribution, the hardware filter was used in combination with the *shunt_index* to shunt parts of the traffic to the software. There were no rules loaded into `netfilter`. All incoming packets are forwarded by the Linux bridge.

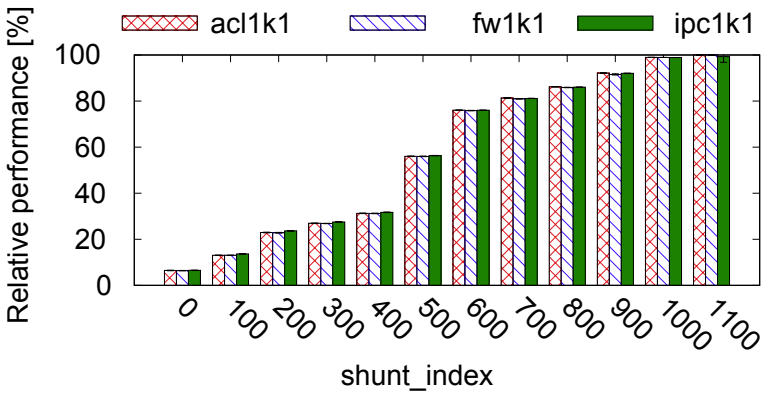


Fig. 3.15: Packet rate of the underlying architecture as a function of the fraction of packets forwarded through hardware. At *shunt_index* = 0 all packets are shunted to the software, while at *shunt_index* = 1100 all packets are forwarded.

We compared the number of ingress packets vs. packets received at the receiver, which is the inverse of packets being dropped in the firewall. Each data point shows the average packet rate of ten 20 second test runs, with distribution of the workload being set by the *shunt_index* as the variable parameter. The average number of ingress packets arriving at the HyPaFilter network interface in each test run before any classification is $22\,945\,747$ ($\sigma = 230\,847$) in 20 s. Figure 3.15 shows the percentage of packets arriving at the receiver. The standard deviation is too small to be visible in the plot. The architecture of the host system and the NetFPGA SUME used as a simple NIC are only capable of processing on average 6.4% of the packets which arrive at the NetFPGA input interface. Increasing the *shunt_index* and therefore reducing the fraction of shunted packets increases the overall amount close to 100% when all packets are directly forwarded by the FPGA NIC.

Strategy Comparison

To evaluate the performance of the different strategies described in Section 3.4, the setup was adapted to use firewalling. Starting with a hardware-only scenario, we measured the impact of rule insertions without updating the hardware filter definition. The insertion of rules instead updates the *shunt_index* register on the FPGA. This subsequently causes an increasing amount of packets to be shunted to the software firewall.

We conducted our experiments by following a certain test cycle for all three strategies and repeated them for each of the three sample rule sets `acl1k1`,

`ipc1k1`, and `fw1k1`. During the test cycle, we measured the average packet throughput after iteration n following these steps:

- implement the sample rule set onto the FPGA and set *shunt_index* to match everything in hardware,
- for test run $n = 0$, insert a new rule at position $P_0 = 1100$ and set *shunt_index* $= P_0$,
- for test run n, insert a new rule at position $P_n = 1100 - 100n$ and set *shunt_index* $= P_n$,
- repeat last step until $P_n = 0$.

For the first part of this test, we used the full set strategy and loaded the complete rule set in `netfilter`. As mentioned in Section 3.4, this leads to a high redundancy in the matching. For example, an update at position index $P = 500$ sets *shunt_index* $= 500$, therefore all packets with *match_index* ≥ 500 will be shunted. These packets will, however, never match the first 500 rules in `netfilter` (counting from index zero), making them essentially useless.

For large *shunt_index*es, the tests confirmed the assumption that significant performance gains can already be achieved by removing the parts of the `netfilter` rule set that correspond to *match_index* $<$ *shunt_index* (cut set strategy).

Figures 3.16, 3.17, and 3.18 show the packet throughput (received packets) for complex string matching rules at different positions. For comparison, the average packet throughput of an equivalent software-only `netfilter` setup is given. The maximum relative packet throughput at `shunt_index` $= 1100$ reaches 30-fold increase. The error bars show the standard deviation.

For the full set strategy, it can be clearly seen that the packet rate behaves non-monotonic and dips near *shunt_index* $= 800$. This can be explained by the combination of two contrary effects: first, with the *shunt_index* decreasing, an increasing number of shunted packets causes the software performance to reach its limit. Second, with the *shunt_index* increasing, the packets that are shunted will only match a decreasingly smaller part at the end of the software rule set. This means that the average number of rules traversed by the packets will also increase, regardless of the constant total number of software rules.

To get a better overview of the performance increase by applying the improved strategies, their packet rate has to be compared against the full set strategy. This

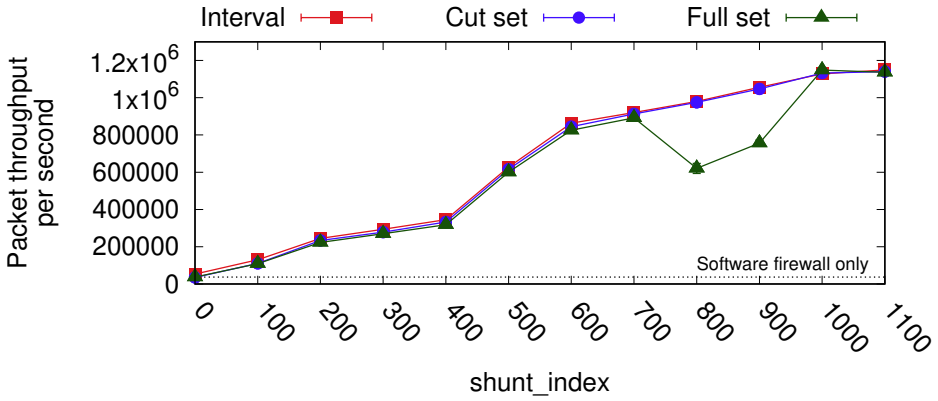


Fig. 3.16: Packet throughput of HyPaFilter with complex inserted rules and different strategies, compared to the average software-only `netfilter` setup: `acl1k1` rule set.

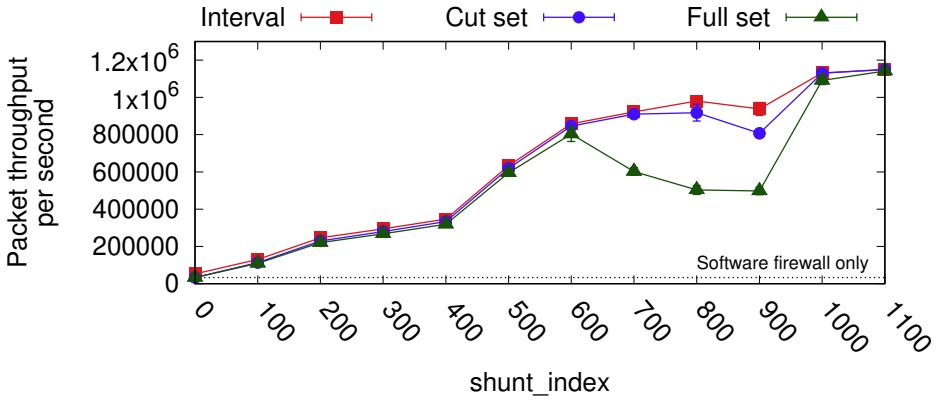


Fig. 3.17: Packet throughput of HyPaFilter with complex inserted rules and different strategies, compared to the average software-only `netfilter` setup: `fw1k1` rule set.

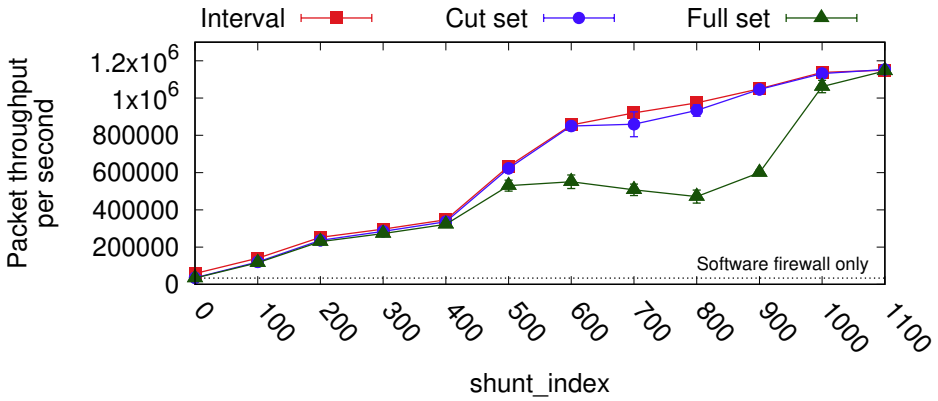


Fig. 3.18: Packet throughput of HyPaFilter with complex inserted rules and different strategies, compared to the average software-only `netfilter` setup: `ipc1k1` rule set.

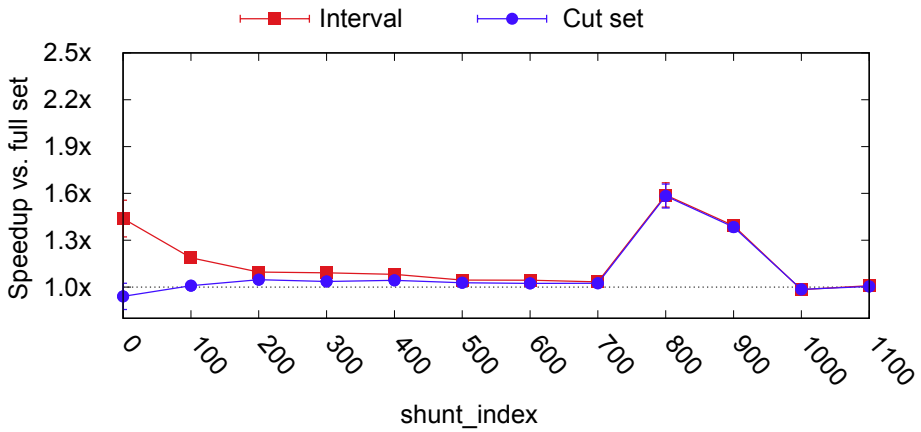


Fig. 3.19: Speedup of the enhanced strategies with complex inserted rules, compared to the full set strategy: `acl1k1` rule set.

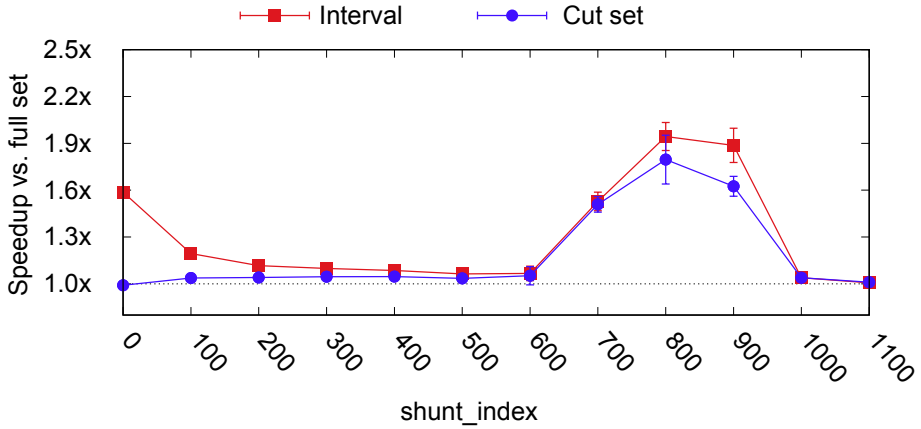


Fig. 3.20: Speedup of the enhanced strategies with complex inserted rules, compared to the full set strategy: `fw1k1` rule set.

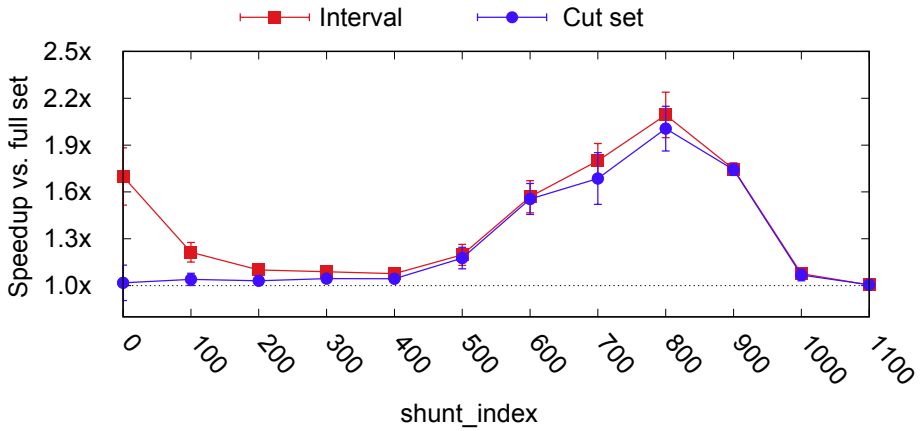


Fig. 3.21: Speedup of the enhanced strategies with complex inserted rules, compared to the full set strategy: `ipc1k1` rule set.

relative speedup, again for using complex string matching rules in the insertion process, can be seen in Figures 3.19, 3.20, and 3.21. Large gains of performance of both the cut set and interval strategy can be seen due to the reduction of the long path effect which is causing the equivalent dip for with the full set strategy near $shunt\_index = 800$. Slight breakouts below the value of one lie within the standard error for the affected data points. With an increased amount of complex software rules with high priority (low *shunt_index*), the advantage of our hardware assisted binary search algorithm used in the interval strategy becomes clear.

3.6.4 Network Latency

While the packet classification rate is the most interesting parameter to measure for evaluation, the additional latency which is added by security appliances can be a major issue for certain applications, e. g., in data centers [114]. Our network latency measurement splits into two parts: the additional delay of the HyPaFilter hardware core in the NetFPGA SUME pipeline, and the actual delay which can be seen on network packets.

The internal additional delay in the FPGA could be determined in the Vivado Simulator and is fully deterministic at 24 clock cycles. With a clock rate of 180 MHz, the core therefore adds an additional delay of 133 ns compared to the NetFPGA SUME in NIC operation.

In order to check for the overall network latency imposed by the HyPaFilter system, the *round-trip time* (RTT) was measured with `ping`, sending 50 packets per test. While a direct connection between sender and receiver (without the NetFPGA SUME) shows an average one way latency of 51 μs ($\sigma = 3.2\,\mu s$), with the HyPaFilter system present and forwarded packets only we saw a tolerable increase to 52 μs ($\sigma = 5.4\,\mu s$). For packets shunted through software without any firewall interaction it further increased to 73 μs ($\sigma = 3.5\,\mu s$). The highest average delay of 96 μs ($\sigma = 7\,\mu s$) occurred with shunted packets and an active software rule set of 1100 rules loaded into `netfilter`.

With the limitation of the uncertainty of the measurement method, the results show that our hardware filtering algorithm is suitable for low latency requirements.

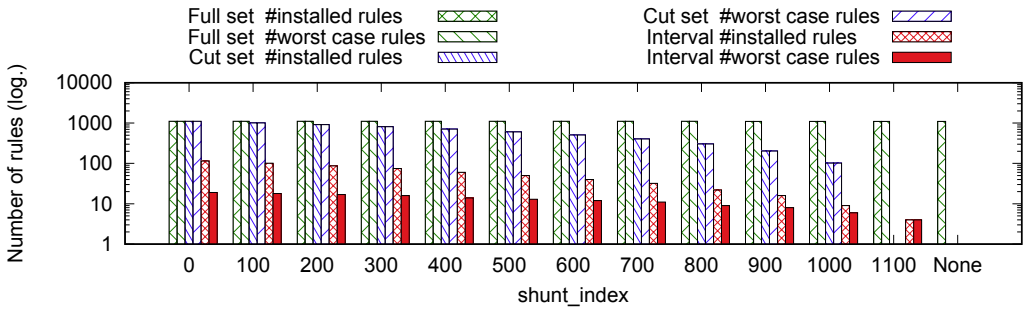


Fig. 3.22: Number of installed/worst case traversed rules for different strategies.

3.6.5 Rule Set Parameters

The strong influence of the number of rules in a software firewall on its classification performance leads to the question how many rules are loaded into the firewall for the three different strategies after applying the update cycle. These numbers were determined by exporting the rules with `iptables-save` and counting the correspondent lines. As HyPaFilter uses a binary tree searching algorithm, we also evaluated the worst case path length, i. e., the highest number of potentially traversed rules for incoming packets. Figure 3.22 gives an overview of the actual number of rules which are active in `netfilter` for different strategies, as well as the number of rules which have to be evaluated in the worst case.

3.6.6 Update Delay

Another interesting parameter is the time required for different types of updates required for different strategies. We therefore measured the time for inserting rules, updating the *shunt_index* register in the FPGA, and uploading a new configuration to the FPGA. According to our test cycle, the delays for the insertion were determined for consecutive insertions of rules at certain positions, i. e., the test at $shunt\_index = 900$ is executed with the assumption of rules previously inserted at position 1100 and 1000. The update process involves different operations for each strategy:

- for the full set strategy, inserting a single rule with `iptables` and setting *shunt_index*,
- for the cut set strategy, truncating the rule set, inserting and loading this set with `iptables-restore`, and setting *shunt_index*,
- for the interval strategy, calculating intervals, inserting the chained rule set with `iptables-restore`, updating the driver and setting *shunt_index*.

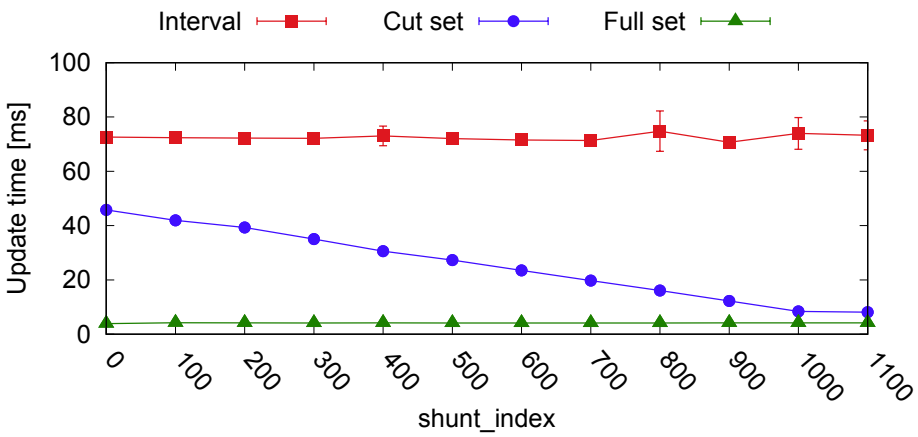


Fig. 3.23: Update latency of different strategies.

Possible discrepancies in the classification result during the update process can be avoided by initially setting the *shunt_index* to 0 and keeping the default drop policy for the software firewall. Figure 3.23 shows the result of this test, as an average of ten test cycles for each data point. Setting the *shunt_index* register on the FPGA alone takes 1 μs. The measured time confirms our assumptions about the cost for rule insertions (see Section 3.4).

The synthesis and implementation process that is used to generate the new bitfile with one of the test rule sets requires about 45 minutes on the described HyPaFilter evaluation host, using Vivado 2014.4. The Xilinx tool `xmd`, which is used to configure the FPGA with this bitfile via the programming interface finishes in 17.38 s. During this time, the network is interrupted. In our test cycles, no hardware update was required to reach the stated results.

3.6.7 OpenFlow SDN

The logical division into a hardware filtering unit with a software backend is in several aspects similar to the concept of an SDN. In a typical SDN switch setup, a controller would place *flows* dynamically into the hardware, allowing fast transmission of matching packets. For a firewall application, the requirements are more complicated than for a simple switch. To build SDN firewalls, controllers with such extended functionality exist [99]. We attempted to compare the performance of HyPaFilter against such an SDN setup.

To this end, the NetFPGA was replaced with a Quanta Computer LB8 48-port SDN [121] switch running PicOS. For a fair comparison, we used a publicly available and stable controller, OpenIRIS v2.2.1 [99], which was installed on the HyPaFilter host system. The OpenFlow 1.3 protocol is used for the communication with the

switch. OpenIRIS includes a firewall module which can be controlled via the REST API [123]. The number of incoming packets was determined through the web interface of OpenIRIS.

We noticed several issues of the OpenIRIS firewall module during our evaluation:

- Rules could not be added to certain positions. Although it is possible to define a `ruleid`, the parameter seems to be ignored and replaced by a random value which is not related to the actual (logical) position of the rule. New rules are always prepended to the current rule set.
- The source and destination port could not be specified as a range.
- The port fields could not be set to values higher than `32767`, obviously due to sign conversion problems.

The update process therefore has to be carried out by first deleting all rules and then adding all rules of our rule set in reverse order. Loading 1100 rules into the module with the REST API takes 3.9 s on average. During our evaluation we found out that the firewall only placed flows into the hardware for ICMP ping packets and established TCP sessions. Our test data (UDP packets), as well as generic TCP packets do not trigger this mechanism, therefore forcing each packet into the slow path to the controller. Although not configurable, this behaviour may be a protection against SYN flooding of the flow table, i. e., purposefully trying to fill the flow table with useless entries. We concluded that due to these effects, a fair comparison against our setup was not possible. These issues were not further investigated.

Apart from this naive standard approach, a more sophisticated way of setting up a hybrid SDN/software firewall is described in Chapter 6.

3.7 Summary

In this chapter we introduced HyPaFilter, a hybrid packet classification approach which combines the parallel matching capabilities of specialized hardware with the extensive matching semantics of widely used software packet filters. HyPaFilter accomplishes this task by partitioning the implemented packet processing policy into a simple and a complex part, where the simple part can be handled directly in hardware and the complex part is installed in the software filter. In-

coming network packets are first processed in hardware and are shunted to the software filter only in the case where complex processing is required. We present a novel strategy how the software-implemented part of the rule set can be organized in order to reuse matching information from the hardware. This strategy can be used on top of `netfilter` and does not require changes of the `netfilter` source code. The actual hardware filter is not limited to our evaluation example. It can be any suitable algorithm which provides the match index. Our evaluation of HyPaFilter based on a combination of a NetFPGA SUME device and a Linux host system demonstrates significant increases in the achievable throughput over a software-only approach, even with rule set constellations where the majority of incoming packets must be processed in software. A major potential for further performance increase could be exploited by handling a larger share of the traffic on the FPGA without involving the software firewall. This will be addressed in the following chapter.

4 HyPaFilter+: Mastering Rule Set Dependencies

This chapter is based on previous work by the author [2, 5]. The crucial improvement discussed in Section 4.3 was the result of a joint project and should also be attributed to the co-authors. The extension of the software implementation was done by Sven Hager.

4.1 Overview

In the previous chapter, we described approaches for building a hybrid FPGA-software firewall classification system. A significant performance increase was achieved by re-using hardware classification data in the software classification process. However, the shunting decision (index-based shunting) follows a straightforward approach by using the index of the most highly prioritized complex or modified rule as the threshold. Each packet that hits a rule beyond this *shunt_index* is shunted to the software firewall, where optimizations mitigate the software classification effort. Nonetheless, the shunting of a packet itself is a severe bottleneck: the evaluation results in Section 3.6.2 show that with all packets shunted, the hybrid system only achieves about 6% of the hardware-only packet throughput. In order to achieve higher throughput gains, especially with more demanding rule set constellations, it is desirable to keep as much traffic on the FPGA as possible. This leads to the question how we can define more precisely which traffic must be shunted.

In this chapter, we therefore describe an analysis method for the rule set that allows to further dissect whether simple rules actually have a logical dependency with regard to the defined checks in the header to a complex rule. For this, we make use of the geometric representation of those checks which will be introduced in the following section. Using this representation, we can apply *header space analysis* (HSA), which was introduced in [50] as a measure to analyse complex firewall rule sets for configuration errors. A similar analysis approach used for anomaly analysis in distributed firewall systems has also been shown in [34].

The extended HyPaFilter approach is called HyPaFilter+, where HSA is used to check whether a complex rule has no logical dependency with a lower prioritized rule. If this is the case, those independent, non-complex rules do not need to be set as shunting rules, hereby reducing shunting traffic. As this requires a support for selective flagging and shunting decisions, the hardware filtering circuit of Chapter 3 needs to be adapted. The evaluation results show that both real and

synthetic rule sets have few logical dependencies among their rules. HyPaFilter+ can therefore sustain the up to 30-fold performance increase significantly longer and for more demanding rule set constellations. In combination, both software and hardware optimization approaches achieve a fast and versatile hybrid FPGA firewall.

4.2 Geometric Representation of Rules

In order to apply an HSA to rule sets, the rule representation of Section 2.2 needs to be extended. Recall that every rule R_i consists of K *checks* $C_i^j : D_j \rightarrow \{\text{true}, \text{false}\}$. Here, the checks C_i^j are assumed to be equality, range, or subnet tests, which are the most common types of tests used in rule sets [38, 54]. Every check C_i^j can be represented as an interval test $H_j \in \left[X_i^j, Y_i^j\right]$, with $X_i^j \leq Y_i^j$ and $X_i^j, Y_i^j \in D_j$. Accordingly, every rule R_i has a geometric representation $G(R_i) \subseteq \mathcal{U}$, which is the K-dimensional hypercube $\left[X_i^1, Y_i^1\right] \times \ldots \times \left[X_i^K, Y_i^K\right]$. We say that two rules R_i and R_k $(i \neq k)$ *conflict* iff there is a header tuple $\mathcal{H}$ that matches both rules, i.e., iff

$$\exists \mathcal{H} \in \mathcal{U} : R_i(\mathcal{H}) \wedge R_k(\mathcal{H}),$$

or equivalently, $G(R_i) \cap G(R_k) \neq \emptyset$.

In Section 2.2, the extended packet classification was described with rules containing simple checks C_i as well as complex checks E_i:

$$R_i = C_i^1 \wedge \ldots \wedge C_i^K \wedge E_i.$$

In our setup, only the simple checks C_i are feasible for both hardware and software classification and are therefore used in the geometric representation. Therefore, for each rule R_i, we define the *geometric reduction* R_i^-, which projects R_i to its simple checks C_i^j and removes the complex part E_i. The geometric reduction is of major importance for the proposed packet classification architecture for two reasons: first, if a rule R_i is a simple rule, then we can implement R_i in hardware on the FPGA. Second, if a rule R_i is implemented in software due to being a complex rule, we leverage its geometrical reduction R_i^- in a preprocessing step in order to decide which packets must be shunted to the software packet processor and which packets can be safely handled in hardware only.

```
-d 1.2.3.4/32 -p tcp --dport 80        -j ACCEPT    # Rule R1
#-------------------------------------------------------------------------------
-s 1.2.3.0/24 -p tcp                  -j ACCEPT    # Rule R2
#-------------------------------------------------------------------------------
-d 1.2.3.5/32 -p udp --dport 3306 -m string        # Rule R3
   --string "SELECT" --algo bm        -j DROP      #
#-------------------------------------------------------------------------------
-d 1.2.3.4/32 -p tcp --dport 443       -j ACCEPT    # Rule R4
#-------------------------------------------------------------------------------
-d 1.2.3.6/32 -p udp --dport 53 -m string          # Rule R5
   --hex-string "|11|2|00|" --algo bm -j DROP      #
#-------------------------------------------------------------------------------
-d 1.2.3.6/32 -p udp --dport 53        -j ACCEPT    # Rule R6
```

Listing 4.1: Example rule set $\mathcal{R}$ in `iptables` syntax.

4.3 Selective Shunting

The crucial contribution of HyPaFilter+ is a new shunting algorithm in contrast to index-based shunting, as introduced in Section 3.3. Index-based shunting can lead to situations where packets are shunted to the host system, although they could not match any more highly prioritized complex rules.

For example, a TCP packet that matches on rule R_4 in Listing 4.1 cannot match the complex rule R_3, since R_3 and R_4 are mutually exclusive. Nevertheless, using index-based shunting, the packet would still be shunted to the host, because R_3 is more highly prioritized than R_4. This, in turn, leads to a higher workload on the software classifier and can eventually result in throughput penalties.

In this section, we introduce *selective shunting*, a method to optimize the shunting decisions taken on the FPGA based on a formal HSA. The selective shunting technique leverages the geometric representations (i.e., the header spaces) of rules to safely narrow down the number of ambiguous rules that result in packets being shunted to the host. In consequence, this increases the likelihood of a packet to be processed by the FPGA alone. To this end, we compute a *shunting vector* $V_{\mathcal{R}_S}$ that stores a single *shunt bit* for every simple rule R_j in $\mathcal{R}_S$. In the following, we denote the jth bit in $V_{\mathcal{R}_S}$ by $V_{\mathcal{R}_S}[j]$. Further, we denote the set of complex rules in $\mathcal{R}$ that are more highly prioritized than R_j by Γ_j. A set shunt bit $V_{\mathcal{R}_S}[j]$ indicates that there exists at least one possible packet header that matches both the simple rule R_j and at least one complex rule in Γ_j. Hence, R_j conflicts with at least one rule in Γ_j. Using the geometric representations of rules, the layout of $V_{\mathcal{R}_S}$ can be expressed by

$$V_{\mathcal{R}_S}[j] = \begin{cases} 1, & \text{if } \bigcup_{R \in \Gamma_j} (G(R^-) \cap G(R_j)) \neq \emptyset \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

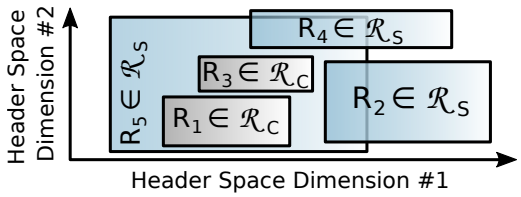


Fig. 4.1: Example sketch for different rules in a reduced two-dimensional header space.

After the shunting vector $V_{\mathcal{R}_S}$ has been computed in a preprocessing step, it is stored on the FPGA. Each time a packet is classified by the FPGA matching circuitry, we use the determined *match_index* in order to look up the shunt bit $V_{\mathcal{R}_S}[match\_index]$. This is in contrast to index-based shunting, where we compared *match_index* with *shunt_index*. Only if the shunt bit is set, the packet is sent to the host for further classification, because the matching simple rule could be overruled by a more highly prioritized complex rule. Otherwise, it can be safely treated entirely in hardware.

We use Figure 4.1 to visualize the difference between index-based and selective shunting in a two-dimensional header space example. With index-based shunting, all packets are shunted due to the complex rule R_1 with the highest priority. In contrast, with selective shunting only those packets are shunted that match the simple rule R_5 on the FPGA, since R_5 is the only simple rule whose geometric shape intersects with those of the complex rules R_1 or R_3.

The procedure SELECTIVE_SHUNTING_ANALYSIS (SSA) for the computation of $V_{\mathcal{R}_S}$ is shown in Algorithm 3. It can be seen that SSA appends one bit to the shunting vector $V_{\mathcal{R}_S}$ for every simple rule in the input rule set $\mathcal{R}$. For each simple rule, the bit is computed by testing whether the intersection of the geometric representation of the simple rule with the geometric representation of any more

Algorithm 3 Compute the shunt vector from the rule set $\mathcal{R}$.

```
1:  function IS_SIMPLE_RULE(Rule R)
2:      return G(R⁻) = G(R)
3:  function IS_COMPLEX_RULE(Rule R)
4:      return G(R⁻) ≠ G(R)
5:  function SELECTIVE_SHUNTING_ANALYSIS(Rule set R)
6:      V_RS ← [ ]   // Initialize V_RS with an empty vector
7:      for i ∈ {1, ..., |R|} do
8:          if IS_SIMPLE_RULE(R[i]) then
9:              bit ← 0
10:             for j ∈ {1, ..., i − 1} do
11:                 if IS_COMPLEX_RULE(R[j]) then // check if rule R[j] is complex
12:                     if G(R[j]⁻) ∩ G(R[i]) ≠ ∅ then
13:                         bit ← 1
14:             V_RS ← V_RS + [bit]    // Append bit to V_RS
15:     return V_RS
```

highly prioritized complex rule in $\mathcal{R}$ is empty. Hence, the runtime complexity of SSA is in $\mathcal{O}\left(|\mathcal{R}|^2\right)$.

For the example rule set $\mathcal{R}$ in Listing 4.1, the shunting vector $V_{\mathcal{R}_S}$ would be computed as follows:

$$V_{\mathcal{R}_S} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{matrix} \text{(for simple rule } R_1\text{)} \\ \text{(for simple rule } R_2\text{)} \\ \text{(for simple rule } R_4\text{)} \\ \text{(for simple rule } R_6\text{)} \end{matrix} \tag{4.2}$$

The rules R_1 and R_2, which are both simple rules, are the first and second rule in $\mathcal{R}$. Hence, they cannot conflict with any more highly prioritized complex rules, and therefore, their corresponding shunt bits are zero. Since rule R_4 does not conflict with the complex rule R_3 due to the different transport layer protocol check, R_4's shunt bit is also set to zero. Finally, R_6's shunt bit is set to one, because it conflicts with the more highly prioritized complex rule R_5. The resulting shunting vector [0,0,0,1] leads to fewer packet shunts than index-based shunting, since packets that first match rule R_4 can be processed entirely in hardware.

We now prove both the correctness and the *HSA-ideality* of shunting vectors computed by the SSA procedure.

Definition 1. ***(False negative)*** *A shunt bit $b = V_{\mathcal{R}_S}[\mathcal{R}_S(i)]$ that corresponds to the simple rule R_i in $\mathcal{R}$ is false negative if $b = 0$ and if there exists a more highly prioritized complex rule R_j with $j < i$ in $\mathcal{R}$ that conflicts with R.*

Definition 2. ***(Correctness)*** *The shunting vector $V_{\mathcal{R}_S}$ for the rule set $\mathcal{R}$ is correct if it does not contain false negative shunt bits.*

Theorem 1. *The application of SSA always results in correct shunting vectors.*

Proof. Theorem 1 follows directly from Algorithm 3: for every bit $b = V_{\mathcal{R}_S}[\mathcal{R}_S(i)]$ that corresponds to the simple rule R_i, every more highly prioritized complex rule R_j with $j < i$ in $\mathcal{R}$ is checked whether it conflicts with R_i. If such a complex rule exists, b is set to 1 and thus cannot be a false negative. □

Definition 3. ***(False positive)*** *A shunt bit $b = V_{\mathcal{R}_S}[\mathcal{R}_S(i)]$ that corresponds to the simple rule $R_i \in \mathcal{R}$ is false positive if $b = 1$ and if there does not exist a more highly prioritized complex rule R_j with $j < i$ in $\mathcal{R}$ that conflicts with R_i.*

Definition 4. ***(HSA-ideal shunting vector)*** *An HSA-ideal shunting vector $V_{\mathcal{R}_S}$ for the rule set $\mathcal{R}$ is correct and does not contain any false positive shunt bits.*

Theorem 2. *For any rule set $\mathcal{R}$, the application of SSA always computes an HSA-ideal shunting vector.*

Proof. Let $V_{\mathcal{R}_S}$ be the SSA-computed shunting vector for a given rule set $\mathcal{R}$. The correctness of $V_{\mathcal{R}_S}$ follows from Theorem 1. Assume that $V_{\mathcal{R}_S}$ contains a false positive shunt bit $b = V_{\mathcal{R}_S}[\mathcal{R}_S(i)]$ for the simple rule R_i. Since b was set to 1 by the SSA procedure, there must exist an index j with $j < i$, such that the rule R_j is complex and conflicts with R_i. This contradicts the assumption that b is false positive. □

We call a simple rule $R_i \in \mathcal{R}$ a *shunting rule* if the employed shunting technique (i.e., index-based or selective shunting) decides that a packet with *match_index* $= \mathcal{R}_S(i)$ must be shunted to the host for further processing. If this is not the case, we call R_i a *non-shunting rule*. Intuitively, it is desirable that many simple rules in $\mathcal{R}$ are non-shunting rules, as this will allow hardware-only processing if these rules are the highest prioritized matching ones.

In comparison to index-based shunting, selective shunting requires higher preprocessing times in case of a rule set update in order to compute the shunt vector. However, this effort is rewarded by possibly much fewer packets having to be shunted to the host at runtime. This, in turn, leads to significantly higher packet processing rates, as we will show in our evaluation.

We point out that it is possible to even further reduce the number of shunted packets by installing the geometric reduction R_i^- of every complex rule R_i in the hardware matcher. In this case, the size of the hardware-computed result vector V_{res} increases from $|\mathcal{R}_S|$ to $|\mathcal{R}|$, as the hardware matcher also generates one result bit for the geometric reduction of every complex rule. Consequently, a packet P only has to be shunted if the most significant set bit $V_{\text{res}}[i^*]$ corresponds to a complex rule $R_{i^*} \in \mathcal{R}$. This approach requires modifications of our rule set-tailored matching circuitry in case of a rule set update, which is very time-consuming. In contrast, a software rule set update could be carried out in less than two seconds in most cases. Hence, we opted for the proposed shunting strategies. A hybrid approach exploiting this possibility by using dynamic update capabilities will be treated in Chapter 6. For stateless rules, the analysis could also further evaluate whether the action of a complex rule and a conflicting simple rule is equivalent. In this case, the simple rule does not need to be set as a shunting rule.

```
-d 192.0.2.0/24 -s <EXTNET> -p tcp -m conntrack                    # R1
  --ctstate ESTABLISHED -j ACCEPT                                  #
#-------------------------------------------------------------------------
-s 192.0.2.0/24 -d <EXTNET> -p tcp -m conntrack                    # R2
  --ctstate NEW          -j ACCEPT                                 #
```

Listing 4.2: Example stateful rule set $\mathcal{R}_A$.

```
-d 192.0.2.0/24 -s <EXTNET> -p tcp -m conntrack                    # R1
  --ctstate ESTABLISHED -j ACCEPT                                  #
#-------------------------------------------------------------------------
-s 192.0.2.0/24 -d <EXTNET> -p tcp -j ACCEPT                       # R2
```

Listing 4.3: Example rule set $\mathcal{R}_B$ without explicit call of the `conntrack` module in R_2.

4.4 Firewall Semantics with Implicit State-Tracking

HyPaFilter+ uses `netfilter/iptables` as the software firewall compartment and also relies on its syntax for describing rule sets. `netfilter` uses an implicit state-tracking mechanism, meaning that any packet handled by the software firewall can alter and affect states, even if it hits a stateless rule or the default policy. A typical example for a stateful firewall policy is given in Listing 4.2.

Rule R_2 allows to establish new, outgoing TCP connections. Afterwards, return traffic referring to one of the established connections can pass in. With `netfilter`, however, every packet that is passed through the firewall is automatically tracked by the state-tracking mechanism, without the need for the parameter `-m conntrack`. This means that packets hitting rule R_2 in the altered example policy in Listing 4.3 will also create and affect connection states that can be referred to by rule R_1. The only difference of the two examples is therefore that R_2 in Listing 4.3 allows any outgoing TCP traffic, not just new connections and does not explicitly specify the use of the `conntrack` module.

This behaviour necessitates special care at the analysis of the rule set if the goal is to mimic an identical semantic for the hybrid system as it would be for a standalone `netfilter` firewall: either, the administrator must ensure that all rules that could possibly affect states for stateful rules contain the correspondent parameter, or the analysis must be extended to respect those dependencies.

This extension can be achieved as follows. First, all traffic that does not match a hardware rule in the FPGA must be passed to the software firewall. This is trivially handled by the default hardware policy. Secondly, for all stateful rules,

Algorithm 4 Compute the shunt vector from the rule set $\mathcal{R}$, preserving implicit state-tracking semantic.

```
1:  function IS_SIMPLE_RULE(Rule R)
2:      return G(R^-) = G(R)

3:  // flips source and destination fields, e.g. IP addresses and port numbers
4:  function REVERSE_DIRECTION(Rule R)
5:      R_rev ← R
6:      R_rev.src ← R.dst
7:      R_rev.dst ← R.src
8:      return R_rev

9:  function SELECTIVE_SHUNTING_ANALYSIS(Rule set R)
10:     V_RS ← [ ]   // Initialize V_RS with an empty vector
11:     for i ∈ {1, ..., |R|} do
12:         if j = i then
13:             continue
14:         if IS_SIMPLE_RULE(R[i]) then
15:             bit ← 0
16:             for j ∈ {1, ..., |R|} do
17:                 if R[j]^- ≠ R[j] then // check if rule R[j] is complex
18:                     if R[j].stateful = false then // rule is complex but not stateful
19:                         if j ≥ i then
20:                             continue
21:                         if G(R[j]^-) ∩ G(R[i]) ≠ ∅ then
22:                             bit ← 1
23:                     else   // rule is complex and stateful
24:                         if G(R[j]^-) ∩ G(R[i]) ≠ ∅ then
25:                             bit ← 1
26:                         if G(R[j]^-) ∩ G(REVERSE_DIRECTION(R[i])) ≠ ∅ then
27:                             bit ← 1
28:             V_RS ← V_RS + [bit]    // Append bit to V_RS
29:     return V_RS
```

the dependency analysis must be expanded to the full rule set, in contrast to only evaluating rules with higher priority. In order to cover return traffic, the reverse direction of the given rule must also be considered, i.e., the analysis must be run with flipped source and destination fields as well. This extended analysis is given by Algorithm 4. As for the shunting analysis described in Section 4.3, all dependent rules must then be set as shunting rules and incorporated in the rule set of the software firewall compartment. Note that the `conntrac` module also supports a referral to *related* connections. This can be used for protocols like, e.g., active mode *File Transfer Protocol* (FTP), where different ports are used for a data and a control channel. This more advanced feature is not covered by the extended analysis.

4.5 Software and Hardware Filter Adaptions

The selective shunting approach requires changes to the hardware classification unit, as the former shunting decision in HyPaFilter was based on index comparison

only. Furthermore, the geometric dependency analysis allows for an improvement to one of the software classification strategies described in Section 3.4.

4.5.1 Software Strategy Improvement

Recall the cut set strategy introduced in Section 3.4. Here, β was defined as an index such that a shunted packet can never match a rule $R_i \in \mathcal{R}$ with $i < \beta$. In the case of index-based shunting, β is equal to *shunt_index*. For selective shunting, β is the index of the most highly prioritized simple rule $R_\beta \in \mathcal{R}$ with $V_{\mathcal{R}_S}[\mathcal{R}_S(\beta)] = 1$. We already know that no simple rule with an index less than β can match a packet that has been forwarded to the software filter—otherwise the packet would have been processed solely on the FPGA. In both cases, the rule set installed in the software filter equals $\mathcal{R}_C$ and in $\{R_j | R_j \in \mathcal{R}_S \wedge j \geq \beta\}$.

With the same HSA calculations that we used in selective shunting, this rule set can be further reduced, since the analysis exactly determines which rules of $\mathcal{R}_S$ the packet could match. Therefore, it is only necessary to install all rules in $\mathcal{R}_C$ and in $\{R_j | R_j \in \mathcal{R}_S \wedge V_{\mathcal{R}_S}[j] = 1\}$ in the software filter. We will call this improved variant *HSA cut set strategy*. This reduction produces correct results, regardless of whether index-based or selective shunting is used in the hardware matching unit.

4.5.2 Hardware Classification Unit

The decision function, which was introduced in Section 3.4 to choose whether to shunt or to forward a packet, was defined as *match_index* $\geq$ *shunt_index*. With selective shunting, this function needs to be replaced by a decision on a per-rule basis: packets are shunted if $V_{\mathcal{R}_S}[\textit{match\_index}] = 1$. The same LLO filter—an MPFC—as for HyPaFilter is used, but directly combined with the shunting vector $V_{\mathcal{R}_S}$. This vector is implemented as a simple bit vector register that can be written at runtime from the host. Nevertheless, any matcher can be used as long as the matching information for all rules—rather than only the one with the highest priority—can be extracted.

4.6 Evaluation

HyPaFilter+ aims to achieve an additional performance increase over HyPaFilter by reducing the number of shunted packets. Therefore, we investigate in our evaluation the extent to which the classification performance of a representative

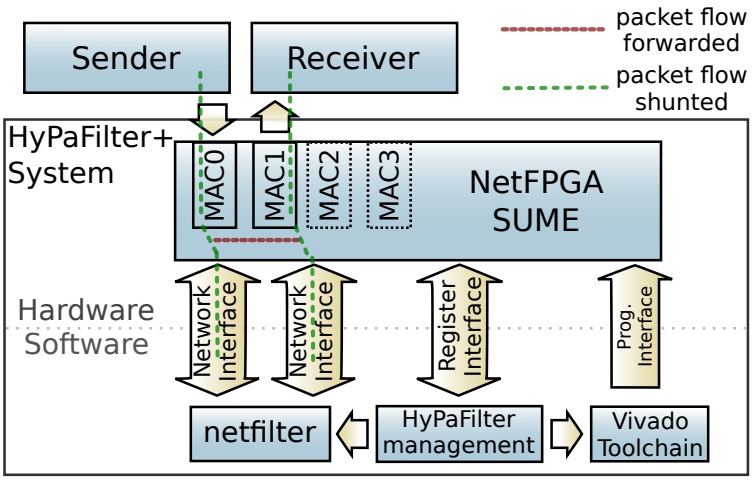


Fig. 4.2: Evaluation setup showing the relevant components. Traffic is generated on the sender and directed through the bridging HyPaFilter+ firewall.

software filter can be improved when used with HyPaFilter+, and how the performance varies with rule set size and structure. Furthermore, we evaluate in detail the impact of newly introduced shunting techniques over the index-based shunting used in HyPaFilter on the number of packets that must be processed on the host system. The measurement setup is similar to the one used for the evaluation of HyPaFilter in Section 3.6, but uses different hardware for the host system, sender and receiver. We used a different test procedure which allows a better evaluation of the performance gains taking into account the new shunting strategy under different conditions. No significant difference was measured for the network latency as conducted in Section 3.6.4, the results are therefore not shown again. Rule set update delays require additional computations when using the HSA for selective shunting.

4.6.1 Test Setup

The measurement setup is similar to the setup used for evaluating HyPaFilter in Section 3.6. A minor difference regards the host, sender and receiver systems. Details of the configuration can be found in Table 4.1. The bridging firewall scenario with the HyPaFilter+ components is shown in Figure 4.2. Further network connections between the systems to remotely start the test cycles and collect the results are not shown.

4.6.2 Test Rule Sets

The rule sets used for the evaluation were identical to those described in Section 3.6.1. In contrast to HyPaFilter, the header fields (IP addresses and port

Sender/Receiver Configuration	
Host OS	CentOS 6.6
NIC	Intel 82599ES Dual 10 Gbit/s
CPU	Intel E3-1270
FPGA Host Configuration	
Host OS	Debian 8.3
NIC	Intel 82599ES 10 Gbit/s
CPU	Intel Xeon E3-1230
FPGA PCIe Card	NetFPGA SUME release 1.0.0
Development Software	Xilinx Vivado 2014.4
Software Firewall	`netfilter` and `iptables` v1.4.21

Tab. 4.1: System configuration.

numbers) of the rules are now decisive for the shunting decision. To better reflect this in our evaluation, we again used the three real world rule sets[1]. As the focus lies on the relationship between the rules with regard to the aforementioned header fields, these rule sets were prepared in a similar way, i.e., set as UDP rules and truncated to the same size as the synthetic rule sets. Further parameters were neglected. Additionally, we created larger rule sets for a scalability test in Section 4.6.7. These rule sets are about the same size as the original real world rule sets.

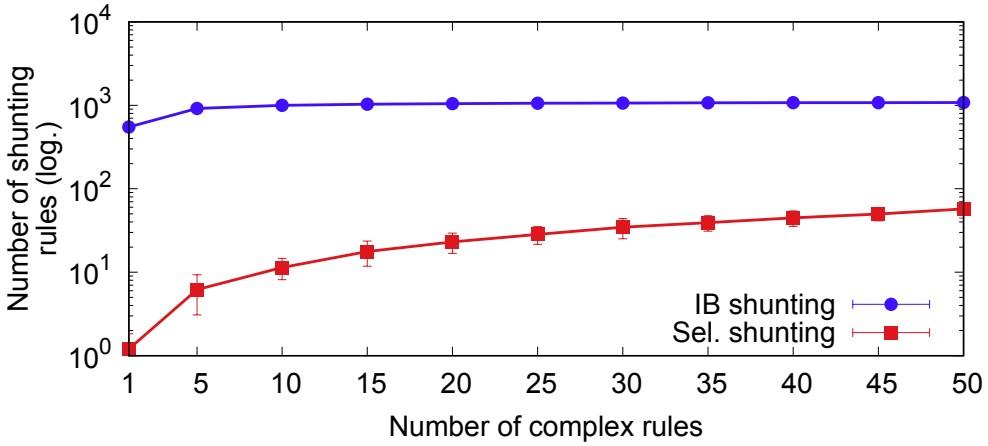
In contrast to the insertion of complex rules at fixed positions with increased priority as used in the evaluation of HyPaFilter, the procedure was adapted to modify the rule set during the test as follows: at k positions ($k \in \{1, 5, 10, 15, \ldots, 50\}$) equally distributed over the rule set (i.e., at $k = 1$ the middle rule), the simple rules at these positions were augmented with the same string matching and probabilistic matching part as used for HyPaFilter. This is intended to show the possibility for demanding worst-case complex rules.

For all rule sets, we used ClassBench's *trace_generator* to generate a trace of $100\,000$ packet headers that will hit the corresponding rule set uniformly distributed, i.e., each rule will be targeted by approximately the same number of packets in the trace.

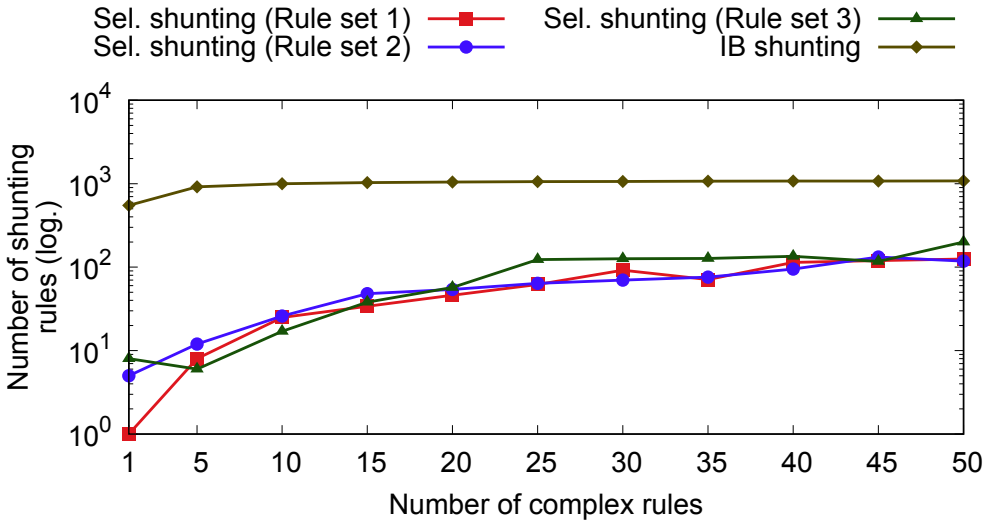
4.6.3 Shunting Technique Effectiveness

In this section, we evaluate the effectiveness of the two shunting techniques introduced in Section 4.3 (using the standard analysis method for selective shunting)

[1] These rule sets were acquired from a customer's firewall setup. For confidentiality reasons, they can unfortunately not be published.



(a) Synthetic ClassBench rule sets (averaged, with standard deviation).



(b) Real rule sets.

Fig. 4.3: Number of shunting rules, index-based (IB) vs. selective (sel.) shunting.

in terms of the number of shunting rules and the number of actually shunted packets. Recall that a packet P is shunted to the software filter if the most highly prioritized matching simple rule is a shunting rule. Hence, if all simple rules in $\mathcal{R}$ are shunting rules, every incoming packet will be processed in software. Likewise, if there are no shunting rules in $\mathcal{R}$, the entire traffic can be processed solely in hardware at line speed. We measured these two quantities by performing the following experiment for every synthetic and real rule set:

1) Load the initial simple rule set.

2) Add complex part to k ($k \in \{1, 5, 10, \ldots, 50\}$) rules at equally-spaced positions from the initial rule set.

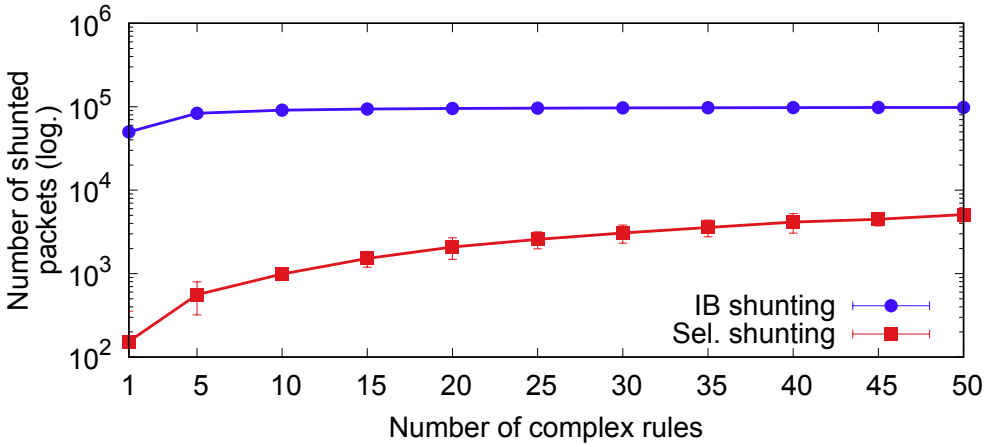
3) Match the corresponding trace file using index-based and selective shunting against the installed rule set.

4) Count the numbers of shunting rules and shunted packets.

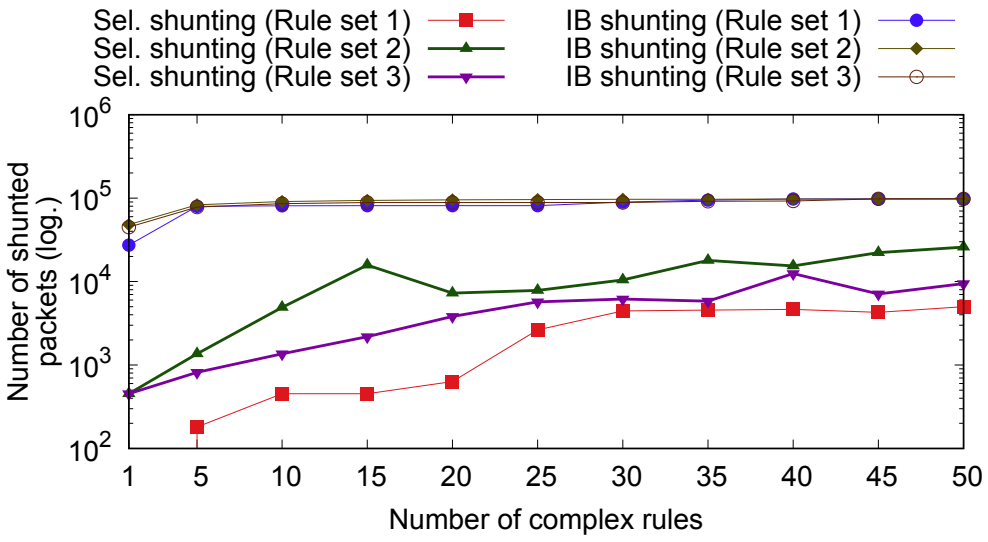
The numbers of shunting rules and shunted packets are shown in Figures 4.3 and 4.4, respectively. It can be seen that the index-based shunting technique results in shunting of a relatively large number of packets—all packets with *match_index* greater than or equal to the lowest modified rule index. Selective shunting reduces the number of shunting rules significantly. Of course, the actual number of selectively shunted packets depends on the rule set characteristics, i.e., the number of simple rules that conflict with more highly prioritized complex rules. Figure 4.3 reveals that ClassBench-generated rule sets are nearly independent regarding header space, leading to an almost equal number of shunting rules compared to the number of complex rules. Hence, selective shunting performs particularly well.

However, the real rule sets are more diverse with regard to their intention and have more header space variation. Depending on which rules are altered and extended by a complex part in the test, this can result in fewer shunting rules, even when more complex rules are used. Nevertheless, also in the case of real-world rule sets, the number of non-shunting rules in case of selective shunting is still at least one order of magnitude greater than in the case of index-based shunting, as shown in Figure 4.3. With index-based shunting, the number of shunting rules is identical for all rule sets, which is why only one line is shown. Since the packets generated by ClassBench's trace generator activate the rules with an approximately uniform distribution, the number of shunted packets shows a distribution that is similar to the number of shunting rules, as confirmed by Figure 4.4. The figure indicates that selective shunting significantly reduces the number of software-processed packets and is therefore superior to index-based shunting with regard to the overall system performance.

The extended dependency analysis described in Section 4.4 ensures the software firewall compartment sees every packet that could possibly affect states that are referred to by stateful firewall rules. This can increase the number of shunting rules, as additional types of dependencies are regarded. The next evaluation will therefore determine and compare the number of shunting rules for both analysis methods. Figure 4.5a shows the result of this test for the synthetic ClassBench rule sets. As seen before, the ClassBench rule sets show few dependencies between the rules. Hence, the extended analysis causes only a minor average increase of the shunting rules. In Figure 4.5b, the test is repeated for all three real rule sets. Here,

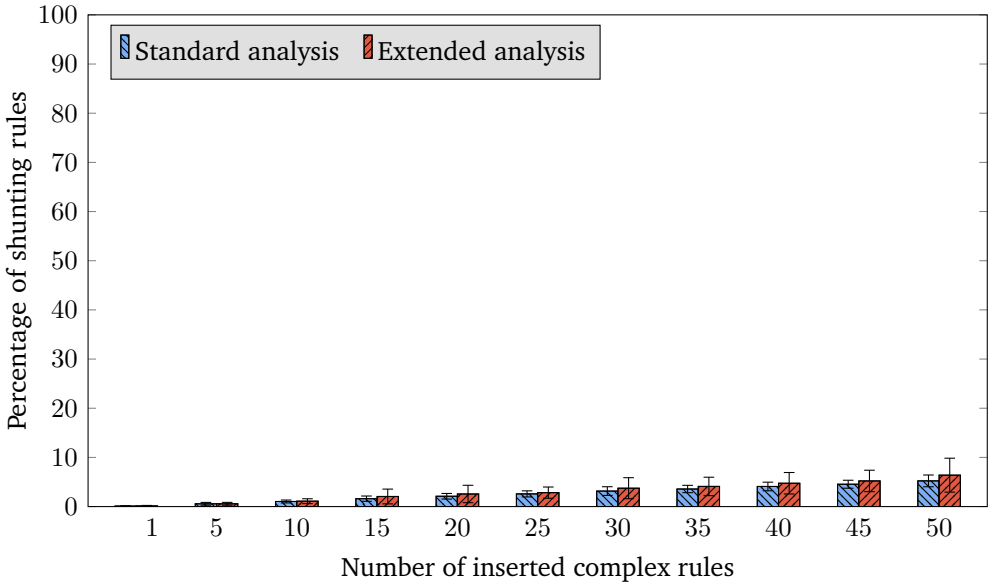


(a) Synthetic ClassBench rule sets (averaged, with std. dev.).

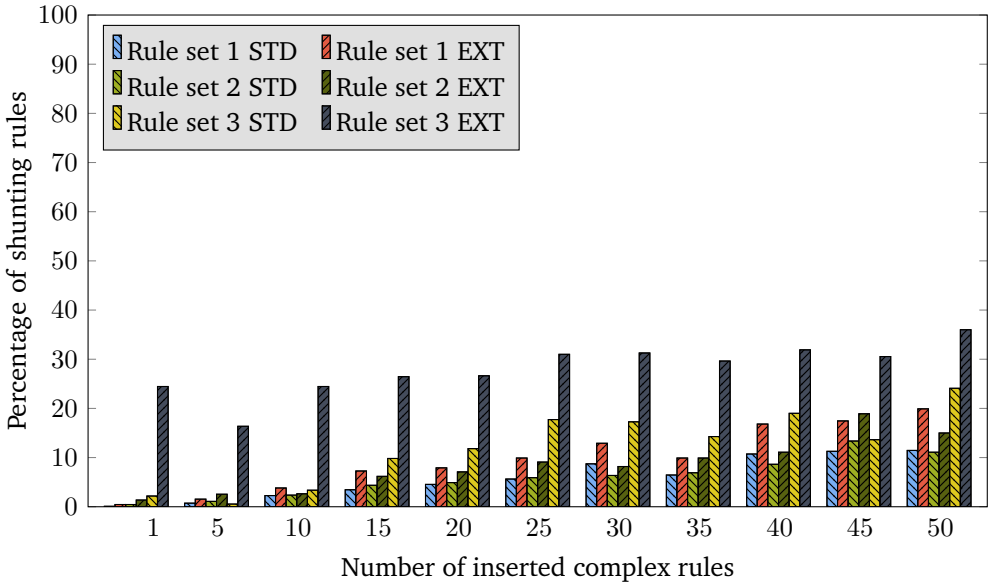


(b) Real rule sets.

Fig. 4.4: Number of shunted packets, index-based (IB) vs. selective (sel.) shunting.



(a) Synthetic ClassBench rule sets (averaged, with standard deviation).



(b) Real rule sets.

Fig. 4.5: Percentage of shunting rules in the hardware filter, with complex rule dependencies analysed agnostic to implicit state tracking (standard, STD) or extended (EXT).

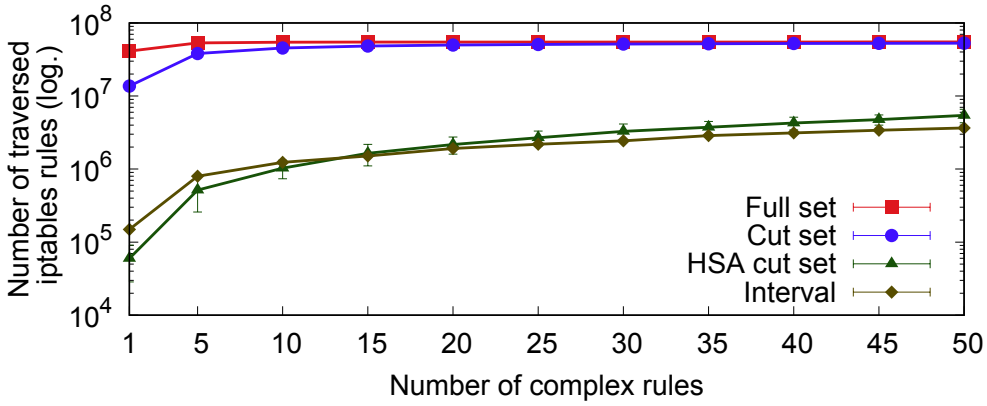
a greater influence of the rule set can be observed. The increase of the number of shunting rules reaches from a minor increase to a significant one of about one order of magnitude. This shows that the full support of this rule set semantic with implicit state tracking can be costly, but still feasible on HyPaFilter+.

4.6.4 Software Strategy Comparison

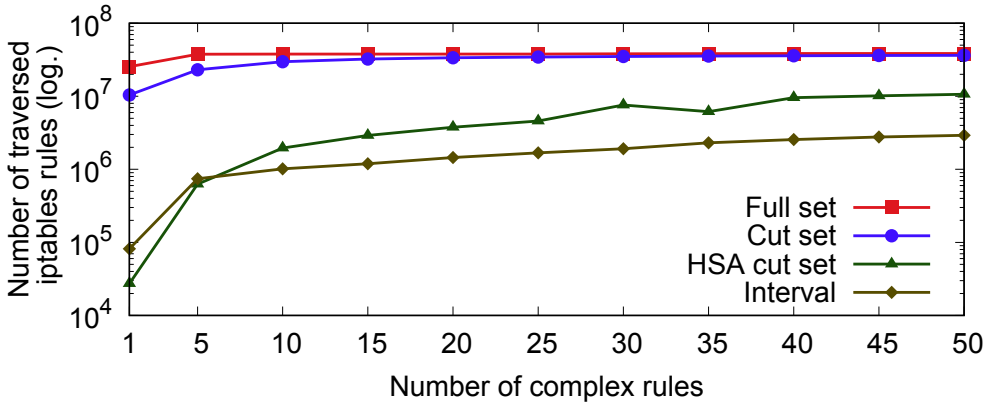
Although the use of shunting strategies, especially of selective shunting, can significantly reduce the workload on the host system, every shunted packet must still be processed in software. Strategies for mitigating the software processing delay and accelerating the classification were already introduced in Section 3.4 and evaluated in Section 3.6. However, as we used different hardware and a different testing procedure, the evaluation was repeated to provide comparable results. We evaluate the strategies in terms of the number of rules that must be traversed to classify all shunted packets. Since the rules are typically traversed linearly in software, this provides a direct indication of the amount of additional work performed on the host system. Here, the examined packets are identical to the shunted packets in Section 4.6.3 using index-based shunting, so the software workload is identical in all test runs. For every shunted packet (for a specific number of complex rules and the selected strategy), we determined the number of rules (the *rule path length*) it traverses in software until it was fully classified. The sums of these path lengths are shown in Figure 4.6.

The average results for the ten synthetic ClassBench rule sets are shown in Figure 4.6a, including their respective standard deviations (using ten test runs).

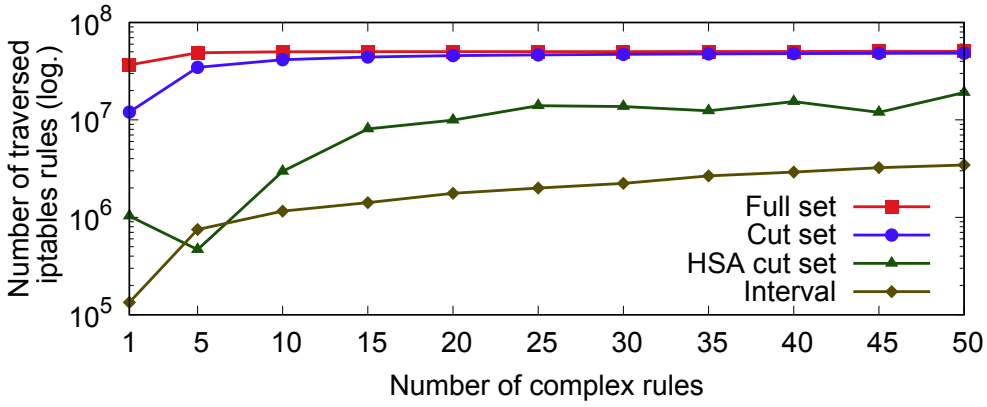
The adapted evaluation setup and test procedure confirms the assumptions and results of Section 3.6.3. In particular, the interval strategy provides the lowest number of traversed software rules for most of the evaluated cases. With real rule sets and more overlap in the header space of the rules, the differences become clearer as it can be seen in Figures 4.6b and 4.6c. Figure 4.6c also emphasizes the different characteristics of the real rules—one single rule may have a significantly greater overlap than n other rules—causing a shorter HSA cut set and therefore less traversed rules at, e.g., five inserted complex rules. These deterministic results clearly demonstrate that the additional processing efforts for the interval or HSA cut set strategies are worthwhile by resulting in significantly lower workload in the software classification engine.



(a) Synthetic ClassBench rule sets (averaged, with standard deviation).



(b) Real rule set 1.



(c) Real rule set 3.

Fig. 4.6: Number of traversed software rules for shunted packets using different strategies. Real rule set 2 omitted as the results are similar to rule set 1.

4.6.5 Packet Rate Measurements

Up to this point, we evaluated the different shunting techniques and approaches to software rule set organization in isolation. In this experiment, we put the pieces together and perform throughput measurements using both index-based shunting and selective shunting (standard analysis) on our hardware platform. We used the interval strategy to organize the rules in the software filter, as this strategy proved to be the most efficient one with respect to classification performance. The measurement method is identical to the packet loss test in Section 3.6.2. We compared the packet throughput rate of HyPaFilter and HyPaFilter+ against a reference measurement of the equivalent software firewalling setup, using `netfilter` and the NetFPGA running as a simple NIC. The procedure for ($k \in \{1, 5, 10, 15, \ldots, 50\}$) is described in the following and was repeated for all test rule sets.

1) Implement the initial test rule set onto the FPGA and set shunting vector to match everything in hardware.

2) Modify k rules at equally-spaced positions from the initial rule set and add the complex part.

3) Update the software filter using the interval strategy.

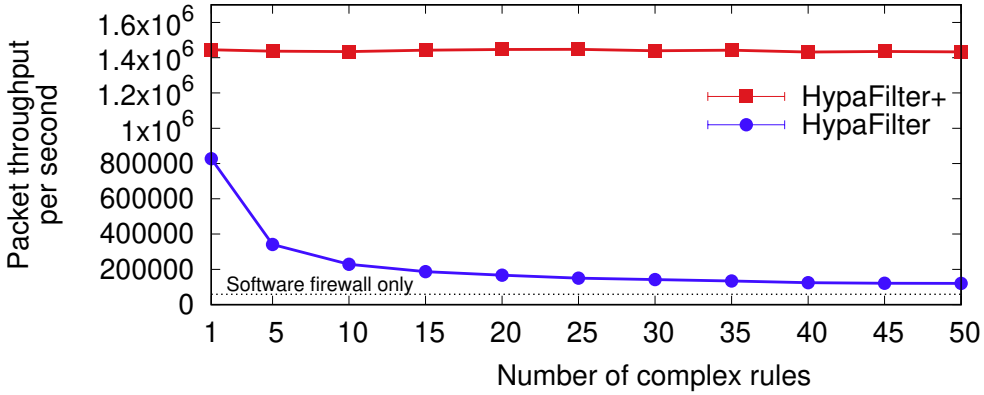
4) Calculate and set shunting registers on the FPGA.

5) Execute the test run.

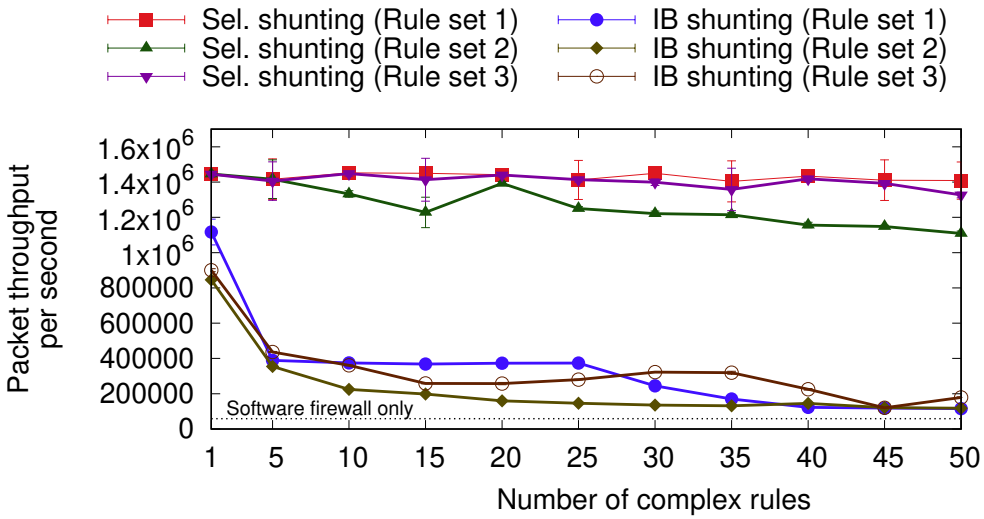
6) Repeat from step 1.

Figure 4.7a shows that the index-based shunting of HyPaFilter—while still faster than software-only—achieves its best results as expected when only a few complex rules are used. This is mainly due to the fact that, with equally-spaced complex rules, an increased number of complex rules results in such rules appearing at higher priorities in the rule set. Therefore, a large amount of traffic is shunted. On the contrary, since only few packets need to be shunted, the selective shunting of HyPaFilter+ is able to maintain a constant packet rate. Compared to the equivalent software-only setup, the results show a 30-fold performance increase.

The results in Figure 4.7b confirm that these numbers are still valid for real rule sets. As explained in Section 4.6.3, in comparison to synthetic rule sets, the real rules have more diverse characteristics. This causes the greater variation and



(a) Synthetic ClassBench rule sets (averaged, with standard deviation).



(b) Real rule sets (averaged over 10 runs, with standard deviation).

Fig. 4.7: Packet throughput of HyPaFilter+ compared to software-only with index-based (IB) and selective (sel.) shunting.

non-monotonic behaviour during this test. As for each k, the same rules are modified and, in contrast to the synthetic rule sets, the results are not averaged, this has therefore no effect on the standard deviation.

4.6.6 Rule Set Update Delay

Another interesting parameter is the time required to update the rule set using the different strategies. We therefore measured the time for modifying rules and updating the *shunt_index* or shunting vector registers on the FPGA. The delays for the insertion were determined for consecutive rule insertions. The software strategy updates were first tested with the inexpensive index-based shunting

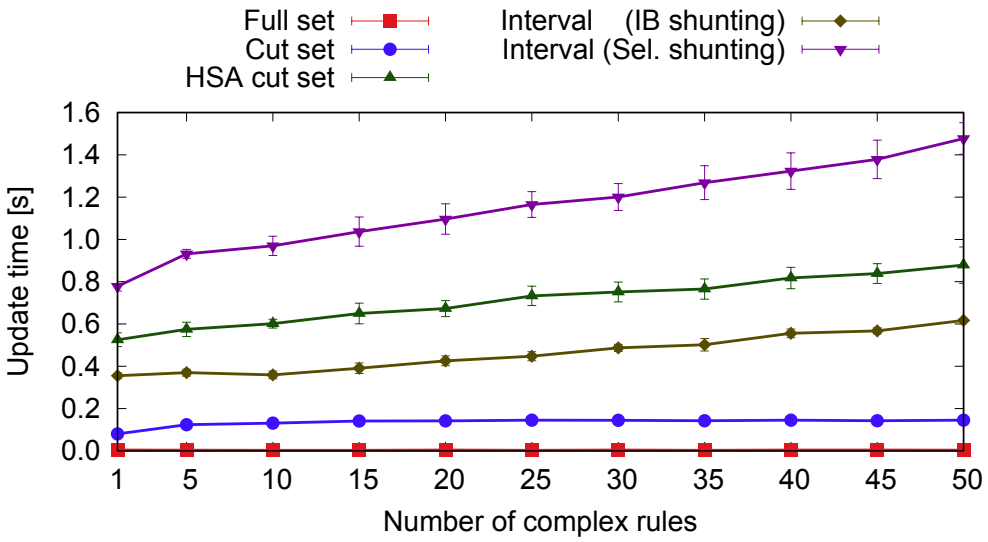


Fig. 4.8: Rule set update latency for consecutive insertion using different software strategies and index-based (IB) or selective (Sel.) shunting (averaged, with standard deviation).

method in order to better distinguish between the delay for the strategies and the calculation of the shunting vector for selective shunting. The following tests have been conducted, taking the time for all steps:

a) for the full set strategy: update a single rule with `iptables` and set *shunt_index*,

b) for the cut set strategy: truncate the rule set, insert and load this set with `iptables-restore`, set *shunt_index*,

c) for the HSA cut set strategy: calculate the shunting vector and create the corresponding rule set, insert and load this set with `iptables-restore`, set the shunting vector registers on the FPGA,

d) for the interval strategy with index-based shunting: calculate intervals, insert the chained rule set with `iptables-restore`, update the driver and set *shunt_index*,

e) for the interval strategy with selective shunting analysis: calculate intervals, insert the chained rule set with `iptables-restore`, update the driver, calculate the shunting vector and set the shunting vector registers.

Figure 4.8 shows the result of this test, as an average of all 13 test rule sets used in the evaluation. Setting one register on the FPGA from the host alone takes 1 μs. This confirms that even the demanding updates of the interval strategy with selective shunting could be carried out with a tolerable delay.

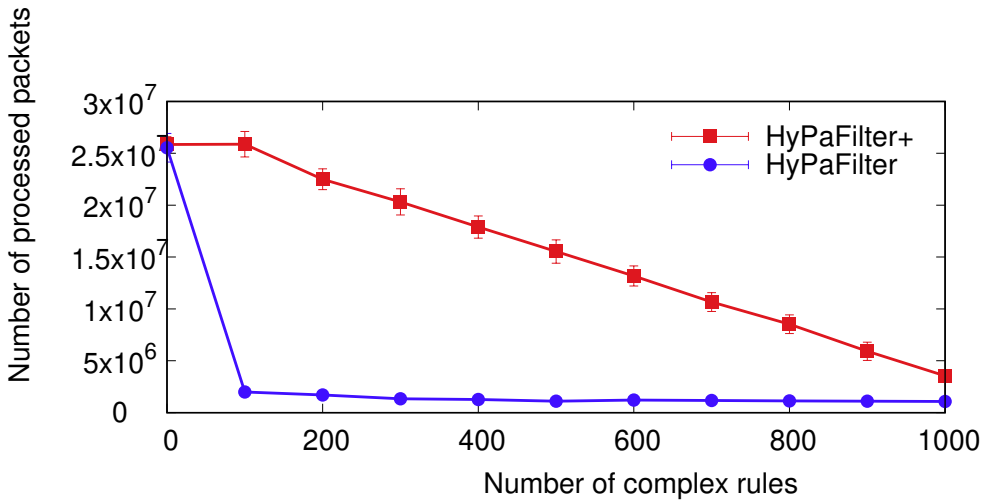


Fig. 4.9: Packet throughput scalability comparison using synthetic ClassBench rule sets of size 1100 (averaged over 10 test runs, with standard deviation).

Updating the logic optimized rule set on the FPGA was only necessary once for every test rule set. Bitfile generation and FPGA programming time took approximately the same time as for HyPaFilter, i.e., 45 minutes and 17 s, respectively.

4.6.7 Scalability

A greater ratio of complex to simple rules will further increase the number of shunting rules, even for the synthetic rule sets with only few dependencies. The possible performance gain of HyPaFilter+ is therefore expected to decrease approximately linearly with an increased ratio. This is confirmed by Figure 4.9, which shows the behaviour for a ratio of up to 1000 complex to 1100 simple rules.

The HyPaFilter+ approach can also be used for larger rule sets than shown so far. The (interchangeable) hardware matching unit can be configured in a pipelined layout which we successfully tested with five stages of 1000 rules each. This increases the latency by 13 clock cycles (72.2 ns at a clock rate of 180 MHz) per stage, without affecting the throughput.

To evaluate the behaviour of the system if a greater number of rules is placed into the FPGA and further, more rules are adapted to require complex processing, we repeated our former tests with 5000 rules and up to 1000 complex rules. In this case, the number of shunting rules is, on average, 68.6% greater than the number of complex rules. As with this setup a larger fraction of the rules are changed to complex rules, the system's performance will approach the software-only firewall

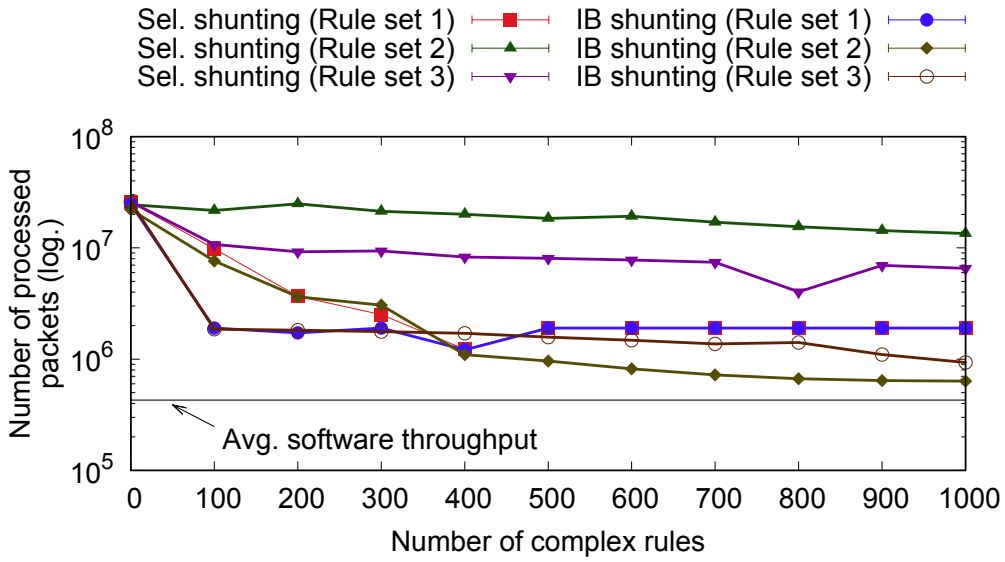


Fig. 4.10: Worst-case scenario speedup using index-based (IB) and selective (sel.) shunting, using real rule sets of size 5000.

level, as shown in Figure 4.10. At 1000 complex out of a total of 5000 rules, the speedup vs. software still reaches $19.7\times$ on average.

Regarding update latency, the critical step is the calculation of the HSA vector, which has a complexity of $\mathcal{O}\left(|\mathcal{R}|^2\right)$. In the worst case, this operation took up to 18.3 s in this evaluation. All other update operations (i.e., registers) scale with $\mathcal{O}(|\mathcal{R}|)$.

4.7 Summary

In this chapter, a major performance-improving extension for HyPaFilter was introduced. The former approach demonstrated an efficient hybrid structure for combining the parallel matching capabilities of specialized hardware with the extensive matching semantics of software packet filters. While HyPaFilter's algorithmic optimization was focussed on speeding up the software classification for shunted packets, HyPaFilter+ aims at additionally reducing the number of shunted packets and hereby reducing the overall software workload. To achieve this goal, we leverage a geometric rule representation to analyse dependencies between rules in order to allow an optimal shunting decision. With this selective shunting strategy, our evaluation demonstrates that also with real-world rule sets and a large number of complex rules, software processing can be avoided for a large share of the traffic. As a result, our HyPaFilter+ prototype based on a combination of a NetFPGA SUME FPGA and a Linux host system demonstrates up

to 30-fold increases in the achievable throughput over a software-only approach. While the maximum performance increase is equal to HyPaFilter, HyPaFilter+ can sustain a significant increase also with demanding rule set constellations.

The pipelined approach offers the possibility to integrate a fast dynamic state table on the FPGA that can be used to directly handle stateful rules in hardware. This way, stateful rules could be handled entirely in hardware, which also avoids the additional workload for the software firewall compartment by the extended dependency analysis. Ideally, this table could be altered by the host system as well. Classification circuits with this capability would further allow to place the geometric reduction of complex rules into the fast hardware classification system, which would likely further improve the result. Furthermore, HyPaFilter+ could be combined with other classification systems, which is also the topic of Chapter 6.

5 Extending and Optimizing FPGA-based Network Processing

5.1 Overview

This chapter is based on previous work by the author [1, 3]. The findings should also be attributed to fellow co-authors. The implementation details in Section 5.2.2 are provided for reference only. They are described in [1] and are part of Sven Hager's work, including the tool `hardbit`. The statistical analysis in 5.3.3 was accomplished by Daniel Loebenberger.

The continuous increase in the performance and capabilites of FPGAs has led to a widespread adaption of FPGAs for numerous applications, with computer networks being just one of them. The challenges and limitations for hardware implementations of network functions have been described in the preceding chapters. Chapters 3 and 4 highlighted how these challenges could be mitigated by a hybrid approach. The results of Chapter 4 further clearly demonstrated the importance of keeping a large share of the workload on the FPGA in order to achieve performance gains. Following these findings, this chapter will identify further tasks that could be implemented in hardware, in particular on FPGAs. As introduced in Chapter 2, hardware components are best suited for tasks that have a high potential for parallelization. In contrast, standard software systems with fast general purpose CPUs perform better whenever random access to large memory or a large number of interdependent operations are required.

In this chapter, we will introduce approaches that are feasible to be implemented on hardware and have the potential to further increase the capabilities of hybrid, FPGA-based firewall systems like HyPaFilter+. First, we focus on FPGA-based packet classification with dynamic rule storage, which is challenging particularly for large rule set capacities [49]. As a compromise, we augment our rule-specific, generated FPGA classification circuit which was used for HyPaFilter+, with a dynamic rule storage. This allows for a trade-off between efficient utilization of the FPGA's resources and the capability for dynamic updates without the need for costly updates of the full FPGA configuration.

Secondly, HyPaFilter+ could benefit from a stateful extension, allowing stateful packet classification in the hardware classification system. Therefore, a central building block for stateful packet classification using hash tables is examined: the hash function. Typical hash functions used for this purpose are highly optimized

for CPU implementation to allow fast calculation on standard software-based systems. On the contrary, these popular hash functions are not necessarily well-suited for hardware implementation [81]. To achieve fast lookups with hash tables, drawbacks with regard to security due to inadequate algorithms are often approved [24, 117, 118]. In this chapter, we will compare different hash functions and propose a solution that is suitable for FPGA implementation and provides a reasonable level of security against certain attacks on hash tables.

5.2 Efficient Classification Circuits for Online Updates

The hybrid firewalling approaches HyPaFilter and HyPaFilter+rely on the software back-end for fast rule set updates, even if the updated rule is in fact a simple rule that could be natively implemented in the fast FPGA classification circuit. Hence, a potential performance improvement could be achieved if such simple rule set updates could be handled by a hardware classification circuit that has the ability for simple online updates. For HyPaFilter and HyPaFilter+, a logic-level optimized MPFC was used instead, as circuits with dynamic update capabilities have limitations with regard to FPGA implementation when a feasible rule set capacity and performance is required. Examples for dynamically updatable circuits include the popular TCAM [73], decision tree, and bit vector searches [10, 36, 54, 71]. Such a circuit allowing dynamic updates can not gain from logic-level optimizations as it is the case for rule-set specific circuitry like the one used in [5]. In comparison to the optimized circuit, such an approach requires more resources per rule and would therefore fit less rules. However, if this circuit only needs to fit updates to simple rules, a significant improvement could be achieved even by an additional small, dynamically updatable classification engine. This is based on the assumption that rule set updates in typical firewall setups are rather selective, leaving a large part of the original rule set unaffected [34].

This motivates an extension of the logic-level optimized circuit of [5] in a hybrid way by an additional, smaller classification unit that can be dynamically updated. In the following, a rule set specific matcher will be called M_{llo} with its corresponding, derived rule set $\mathcal{R}_{\text{llo}}$. Analogously, a generic matcher and its derived rule set will be called M_{gen} and $\mathcal{R}_{\text{gen}}$, respectively. Since this setup is, again, a variation of the hybrid conjunction of different classification engines, similar optimization techniques as introduced in Chapter 4 can be applied. In particular, an HSA-based dependency analysis can be used to avoid redundancies between the classification engines.

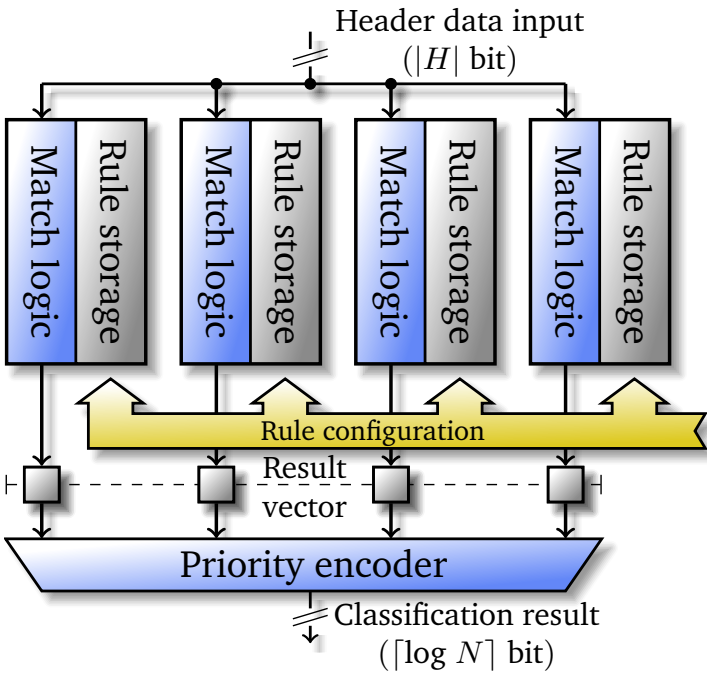


Fig. 5.1: A generic four-element TCAM.

5.2.1 Hardware-centric Classification Circuits

Different hardware-based packet classification approaches have already been introduced in Chapter 2. Here, we will briefly describe the approaches that are used in this section, along with their characteristics. The most prominent and de-facto standard generic classification engine is the TCAM architecture [73]. A TCAM is composed of N parallel match lines. Each line is capable of storing a single rule in a configuration memory, as sketched in Figure 5.1. Incoming packet headers are matched against every line in parallel in a single clock cycle. The result of each match line is stored in a *result vector* at the corresponding position.

FPGA-tailored generic classification architectures are often inspired by decision tree or bit vector search algorithms [33, 49, 72]. These approaches either map decision trees or bit vectors that are generated from the specified rule set onto *static random-access memory* (SRAM) pipeline stages on the FPGA. In comparison to a TCAM, they require more clock cycles for classification, but have lower power dissipation and hardware resource footprints. For example, the StrideBV approach [33] is particularly well suited for FPGA implementation. Its classification operation only requires a small fixed number of SRAM accesses together with a vectorized AND operation per RAM access. Just as a TCAM, StrideBV generates a result vector for an incoming packet header H, but splits the matching process over k pipeline stages. In each stage, $W = \frac{|H|}{k}$ bits of the packet header are used

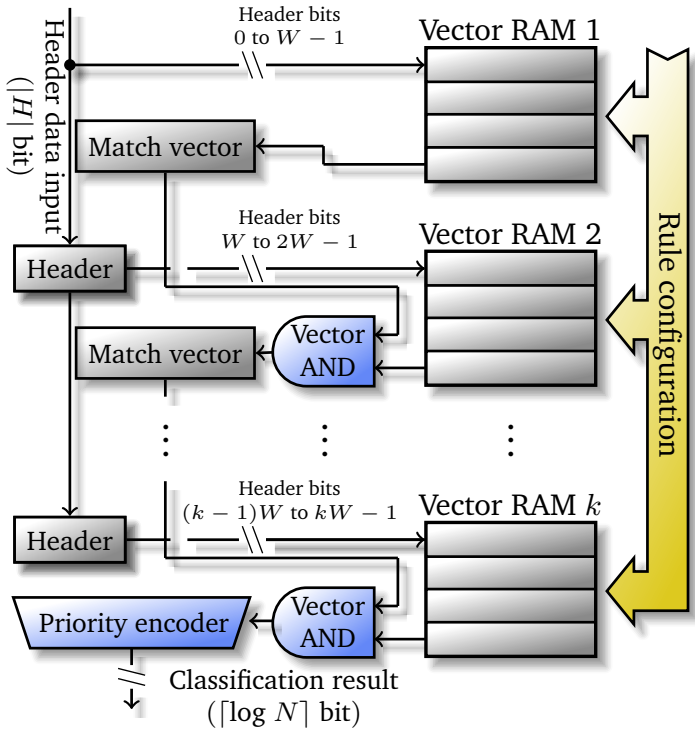


Fig. 5.2: A StrideBV circuit.

as an index for an SRAM that stores a *match vector* for these particular bits. The pipelined vectorized AND of these match vectors finally results in the result vector, as shown in Figure 5.2.

The described approaches are examples for generic matching circuits that store their utilized search data structures in application-level configuration memories. For instance, TCAMs use the rule storages sketched in Figure 5.1, while StrideBV relies on the vector RAMs shown in Figure 5.2. In contrast to dynamic circuits, the actual rule set can directly be evaluated in the implementation process and translated into a specific matching circuit [37, 39, 77]. The MPFC approach [39], which is also used for the hardware classification compartment in the previous chapters, translates each rule in the rule set into a corresponding match circuit, as sketched in Figure 5.3. Note that this circuit is similar to the TCAM in Figure 5.1, with the difference that the MPFC circuit does not have configuration memories, as each match circuit is tailor-made for the corresponding rule. However, this comes at the cost of a time-consuming circuit re-synthesis if the rule set changes.

In the following, we examine and evaluate an hybrid combination of the MPFC with either a TCAM or StrideBV classification unit in order to achieve low power dissipation, a small hardware resource footprint, and quick update capabilities at the same time.

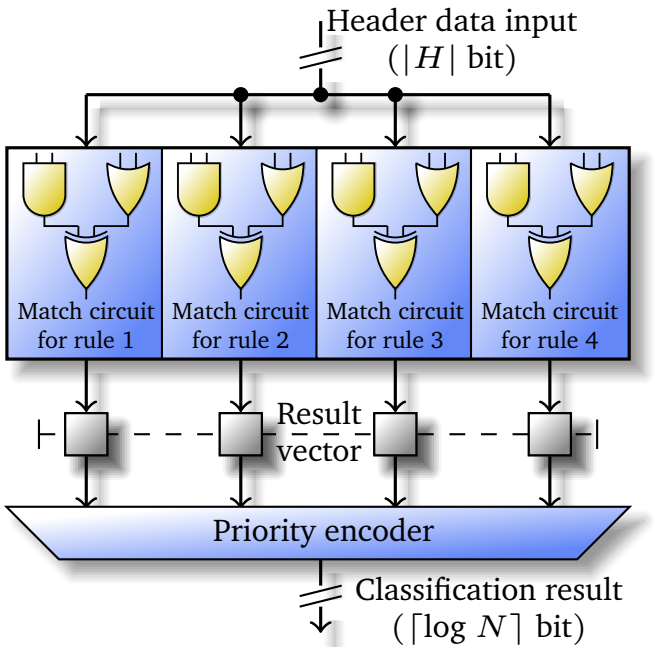


Fig. 5.3: An MPFC match circuit.

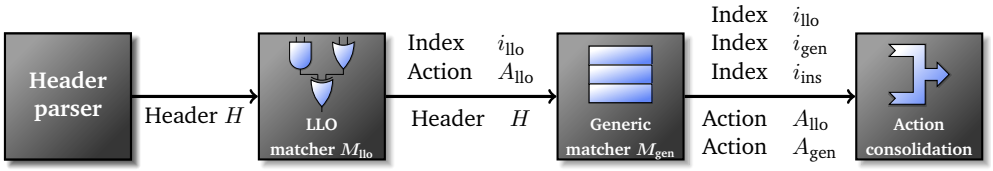


Fig. 5.4: A sketch of the HyPaFilter pipeline with a brief depiction of its components.

5.2.2 Hybrid On-Chip Classification Circuit

The classification pipeline[1] arranges the classification systems M_{llo} and M_{gen} in a processing pipeline, as sketched in Figure 5.4. For each packet, the *header parser* extracts the header fields from the packet and forwards the header tuple H to the first classification system (M_{llo}). As in HyPaFilter, the packet itself is stored in a parallel data pipeline until the action has been determined. The M_{llo} matches the packet against $\mathcal{R}_{\text{llo}}$ in a single clock cycle operation. The result of this step is a preliminary classification result, i.e., the index of the matching rule in $\mathcal{R}_{\text{llo}}$: $i_{\text{llo}} \in \{1, \ldots, |\mathcal{R}_{\text{llo}}|\} \cup \{\epsilon\}$. Moreover, the action A_{llo} of either a matching rule $R^{\text{llo}}_{i_{\text{llo}}}$ or the default policy is forwarded along with the original header tuple H.

To allow updates of the rule set R to take effect without updating M_{llo}, the header tuple H is classified by M_{gen} against $\mathcal{R}_{\text{gen}}$. This classification unit also determines a matching index $i_{\text{gen}} \in \{1, \ldots, |\mathcal{R}_{\text{gen}}|\} \cup \{\epsilon\}$ in $\mathcal{R}_{\text{gen}}$ and the action A_{gen}.

[1] The implementation details in this section are provided for reference only. They are described in [1] and are part of Sven Hager's work, including the tool `hardbit`.

However, this time i_{gen} is not only used to lookup the action A_{gen} from a configuration RAM, but also an *insertion index* i_{ins} that is stored alongside A_{gen}. The insertion index i_{ins} represents the position in the rule set $\mathcal{R}_{\text{llo}}$, from which all rules R_i^{llo} with $i \geq i_{\text{ins}}$ have a lower priority than $R_{i_{\text{gen}}}^{\text{gen}}$, with respect to the total rule set $\mathcal{R}$. Accordingly, we define i_{ins} to be the index of the first rule $R_{i_{\text{ins}}}^{\text{llo}} \in \mathcal{R}_{\text{llo}}$ which is directly behind $R_{i_{\text{gen}}}^{\text{gen}}$ in the total rule set $\mathcal{R}$.

The previously computed matching information A_{llo}, i_{llo}, i_{gen}, i_{ins}, and A_{gen} are then fed to the last step of the classification pipeline, the *action consolidation unit*. Based on the given matching information, this unit acts as an arbiter and decides whether result action A_{res} should be the action A_{llo}, the action A_{gen}, or a default policy. The decision process itself is relatively simple: if neither M_{llo} nor M_{gen} found a matching rule, the default policy is executed. Otherwise, if either M_{llo} or M_{gen} found a matching rule, the action provided by the corresponding matcher is executed. Finally, if both matchers found a matching rule, then the indices i_{llo} and i_{ins} must be compared in order to check whether A_{llo} or A_{gen} takes precedence: if $i_{\text{llo}} < i_{\text{ins}}$, then the overall most highly prioritized rule with regard to $\mathcal{R}$ is $R_{i_{\text{llo}}}^{\text{llo}} \in \mathcal{R}_{\text{llo}}$, and the action A_{llo} must be applied. This is true, because the rule $R_{i_{\text{gen}}}^{\text{gen}} \in \mathcal{R}_{\text{gen}}$ found by the generic matcher is, with respect to $\mathcal{R}$, placed directly in front of the rule $R_{i_{\text{ins}}}^{\text{llo}} \in \mathcal{R}_{\text{llo}}$ with $i_{\text{ins}} > i_{\text{llo}}$. Consequently, $R_{i_{\text{gen}}}^{\text{gen}}$ has a lower priority than $R_{i_{\text{llo}}}^{\text{llo}}$. Following an analogous reasoning, if $i_{\text{llo}} \geq i_{\text{ins}}$, then $R_{i_{\text{gen}}}^{\text{gen}}$ takes precedence over $R_{i_{\text{llo}}}^{\text{llo}}$, and thus the action A_{gen} is chosen. The source code for the pipeline can be generated by the newly written tool `hardbit`, which allows the parametrization of the components.

5.2.3 Implementation Results

The important metrics to evaluate the hybrid on-chip approach are update latency, resource and power consumption. We compare different existing approaches with this regard when implemented on a Virtex-7 690T FPGA. It is the same target platform as it was used in Chapters 3 and 4 and is also based on the same internal data pipeline. Other parameters, such as the clock rate of 180 MHz and the design software Vivado 2014.4 are also identical. Synthesis and implementation steps were evaluated on a CentOS 6 machine with an Intel® Xeon E3-1270 CPU and 16 GB RAM.

As mentioned, one of the main drawbacks of MPFC circuits is the need for re-synthesis for every rule set update. This step takes about 46 minutes and is almost independent from the actual number of rules, as can be seen in Figure 5.5. We use synthetic rule sets with $100, 200, \ldots, 1\,000$ rules generated by the ClassBench benchmark suite [89] which specify source and destination IPv4 addresses, the

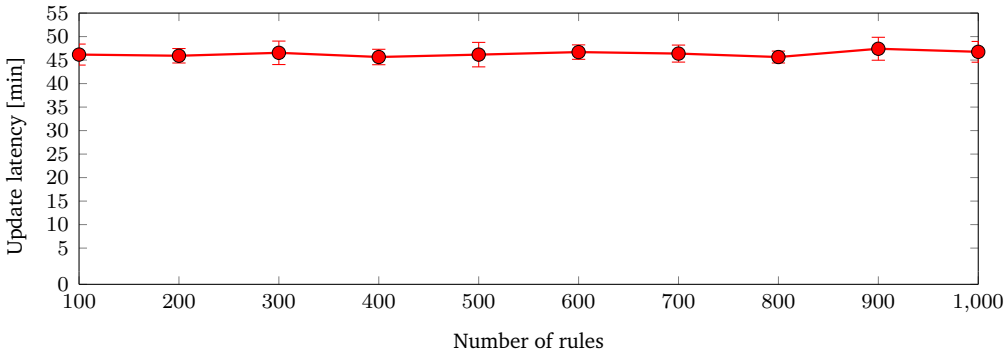


Fig. 5.5: Mean update time for MPFC, with standard deviation.

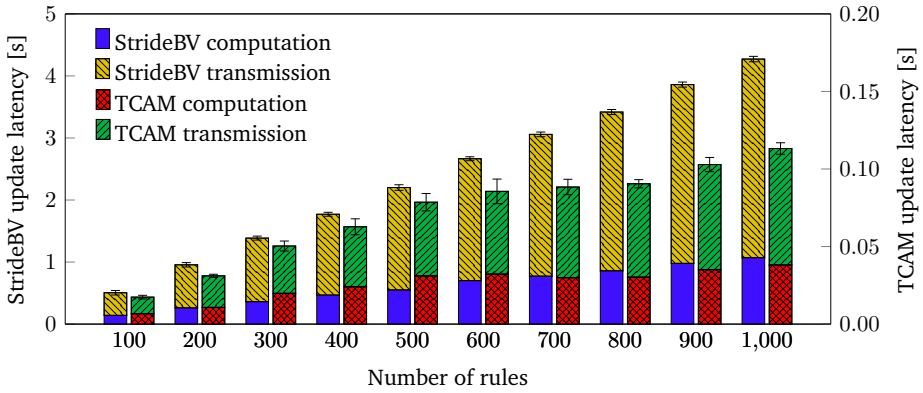


Fig. 5.6: Mean update times (split in computation and FPGA transmission) for StrideBV and TCAM matchers, with standard deviation for total time (note the different y-axes).

transport protocol, as well as source and destination ports. For each rule set size, we generate ten different rule sets.

In order to update the MPFC matcher, three steps are required. Those steps are identical to the initial configuration of the FPGA for HyPaFilter and HyPaFilter+:

1. the software tool translates the rule set into a matching circuit defined in the VHDL,

2. the matching circuit is synthesized, placed, and routed,

3. the generated bitfile is used to configure the FPGA.

This process is time-consuming due to the second step. The VHDL generation can be completed in less than 6 ms and the bitfile transmission finishes in about 17.3 s.

In contrast, updating a generic matcher is significantly faster, as the rule set only has to be translated (*computation*) into the corresponding search data structure, which is subsequently transmitted (*transmission*) via DMA to the StrideBV or TCAM configuration memories on the FPGA. Figure 5.6 confirms that, for the regarded rule sets, rule set updates can be executed in less than 5 s for StrideBV or 0.2 seconds for TCAM, respectively.

Hardware Resource Requirements

In order to investigate the circuit properties of stand-alone MPFC, StrideBV, and TCAM matchers, we insert each of these matchers, one at a time, into the processing pipeline. Subsequently, we synthesize, place, and route the design and finally generate an FPGA configuration bitfile. For each matcher, we examine the number of required FFs, LUTs, and matcher power dissipation for different matcher capacities between $100, 200, \ldots, 1\,000$ rules. In case of the specialized MPFC, we use ten different ClassBench-generated rule sets for each capacity. In contrast, the StrideBV and TCAM matchers must only be built once for each capacity, since their circuitry does not change if their configuration memory contents change. StrideBV can be built based on either distributed RAM or BRAM. The evaluation using distributed RAM proved to be exceptionally resource-intensive, up to designs that could not be implemented by Vivado. For this reason, we are relying on the BRAM variant for our experiments. Furthermore, the TCAM matchers did never meet our timing requirements. The packet processing pipeline uses a single clock domain and a clock source of 180 MHz. Although the design could still be built for some of the configurations, it is therefore not actually usable, as all data propagations rely on correct timing. The TCAM results are still provided for comparison purposes, but we point out that these results should be taken with a grain of salt.

For the rule-set-specific MPFC matcher, the mean values for the ten rule sets for each size are shown, the standard deviation is too small to be visible. In Figure 5.7, we see that the generic StrideBV matcher requires about an order of magnitude more LUTs than the specialized MPFC circuits for every regarded rule set size. Likewise, the TCAM matcher requires an additional order of magnitude of LUTs when compared to StrideBV, as a direct result of their more complex matching circuitry. Figure 5.8 draws a similar picture for the amount of required FFs: MPFC needs the least FFs, the TCAM matchers the most, and StrideBV is in between. The less extreme difference between MPFC and StrideBV can be explained by the fact that StrideBV, in contrast to MPFCs or TCAMs, utilizes additional block RAMs to store its data structure. Finally, Figure 5.9 depicts the power dissipation of the

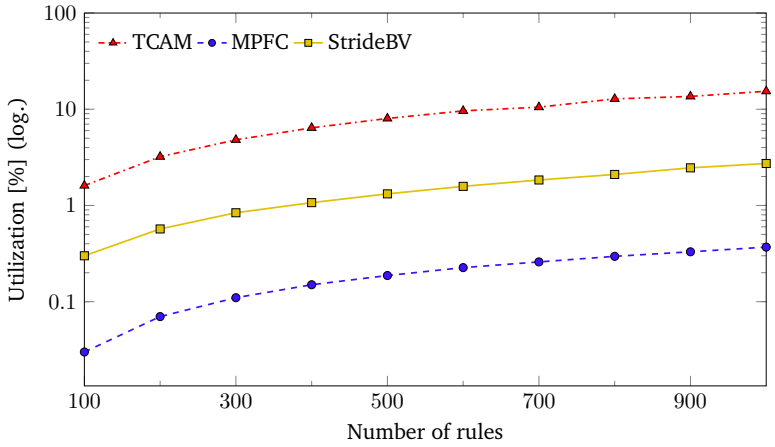


Fig. 5.7: Hardware resource requirements: relative LUT usage.

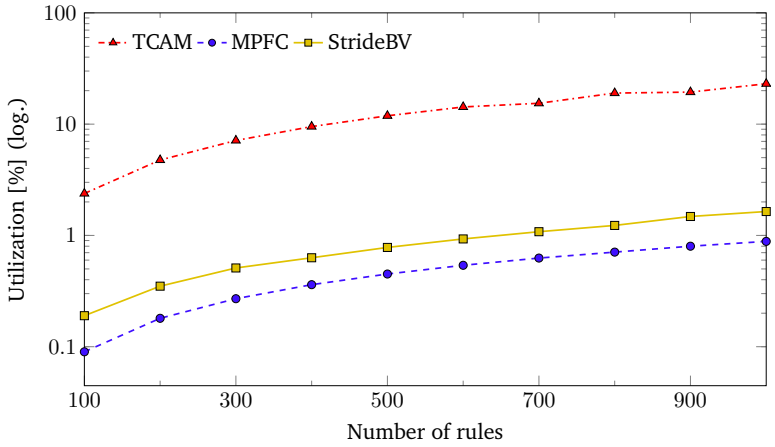


Fig. 5.8: Hardware resource requirements: relative FF usage.

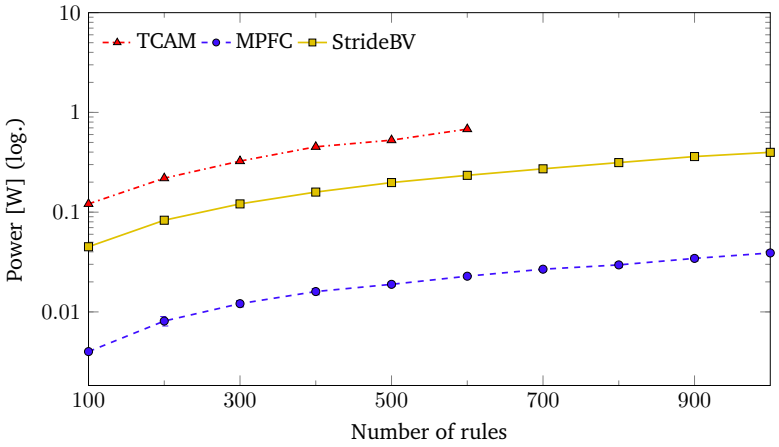


Fig. 5.9: Power dissipation.

three matchers. In case of TCAMs with more than 600 entries, the implementation process of Vivado completely failed to route the design. A reduced clock frequency of 80 MHz instead of 180 MHz allowed this step to at least complete—the final timing constraints were still not met. While this has a negligible effect on the resource requirements, the lower clock frequency drastically reduces the power dissipation since it is directly caused by the process of switching logic gates. Therefore, these data points are not shown. As expected, the MPFC circuit has significantly less power requirements than both the generic StrideBV and TCAM approaches due to its significantly smaller matching circuitry.

As a result summary, Table 5.1 shows the mean hardware resource and power requirements of the three regarded matchers per rule. These values were determined by applying a linear regression on all corresponding data points of the previous test cycle. The results clearly underline that the ability for fast updates, as possible with StrideBV and TCAMs, comes at the price of significantly higher per-rule resource and power usage in comparison with the rule-set-tailored MPFC circuits.

Module	LUTs/rule	FFs/rule	BRAMs/rule	μW/rule
MPFC	2.01	3.46	0.00	37.75
StrideBV	11.65	13.70	0.27	387.00
TCAM	65.82	193.40	0.00	1 075.00

BRAM capacity: 36 Kb

Tab. 5.1: Resource utilization per rule.

Hybrid Configuration

Having seen the significant differences of MPFC, StrideBV, and TCAM with regard to circuit size and power dissipation, we now investigate these properties for a hybrid on-chip matching circuit. To this end, we use MPFC as the M_{llo} matcher and either StrideBV or TCAM as the M_{gen} matcher. We measure LUT usage, FF usage, and power dissipation for the hybrid matcher with $1\,000$ rules, with different distributions of the rules in $\mathcal{R}_{\text{llo}}$ and $\mathcal{R}_{\text{gen}}$. More specifically, we use ClassBench to generate ten different rule sets of each size ($k \in \{0, 100, 200, \ldots, 1\,000\}$) that are used for $\mathcal{R}_{\text{llo}}$. The generic matcher M_{gen} is configured with a capacity of $1\,000 - k$ rules. Subsequently, the matchers are synthesized, placed, and routed by the Vivado design software. For MPFC/TCAM combinations, we experienced the same limitations as mentioned for TCAMs with more than 600 entries and discarded these data points for the power dissipation.

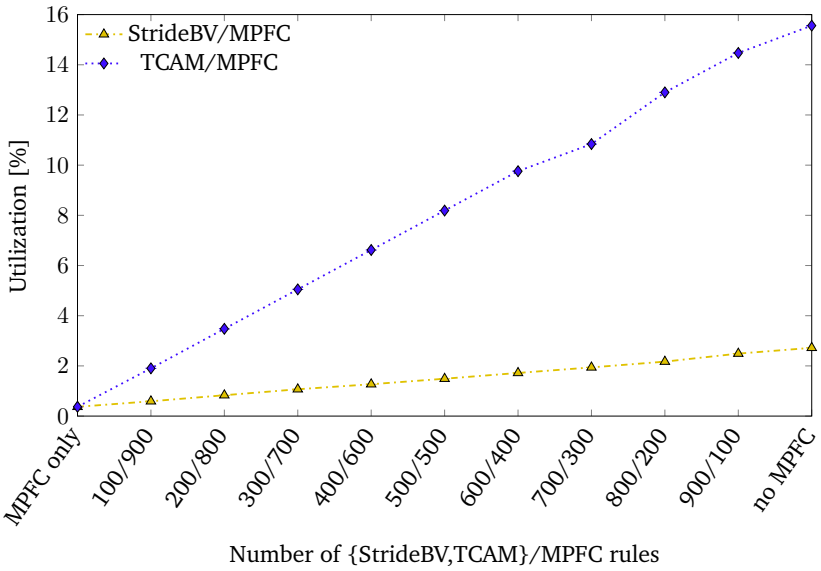


Fig. 5.10: Mean relative LUT usage of hybrid matcher combinations, relative to FPGA resources.

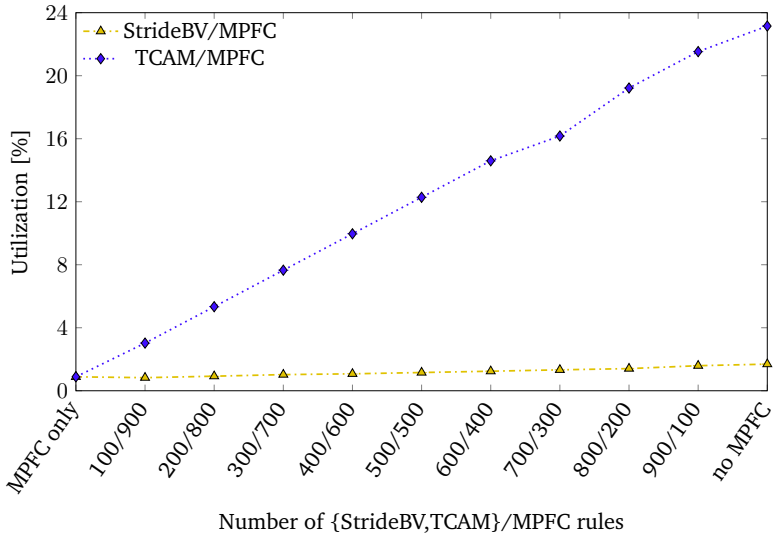


Fig. 5.11: Mean relative FF usage of hybrid matcher combinations, relative to FPGA resources.

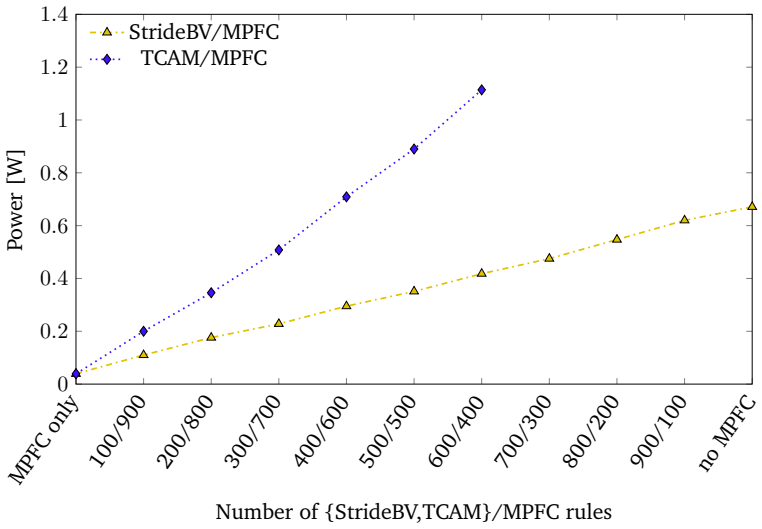


Fig. 5.12: Mean power dissipation of hybrid matcher combinations, relative to FPGA resources.

The mean LUT usage, FF usage, and power dissipation of the hybrid circuits are shown in Figures 5.10, 5.11, and 5.12, respectively. Again, the standard deviations are too small to be visible. In these figures, we observe a linear increase in LUT usage, FF usage, and power dissipation when the generic matcher capacity is increased and the number of rules in the specialized matcher is decreased. Although not shown in the figures, this also holds for the block RAM resources used by the StrideBV/MPFC combination.

These results show that the implementation of a fraction of the rule set in M_{llo} leads to smaller overall circuit sizes and lower power dissipation, when compared to a purely generic matcher. Even when considering a dynamic environment, where 900 of $1\,000$ rules are implemented in the generic matcher and only 100 rules are implemented by the specialized matcher, the hybrid architecture achieves an average reduction factor of $6.3\times$ for LUTs, $15\times$ for FFs and $2.9\times$ for power dissipation compared to a TCAM and about $1.09\times/1.06\times/1.08\times$ when compared to StrideBV. When considering a less dynamic scenario (such as a firewall with many static rules) with a capacity of 200 rules in M_{gen} and 800 rules implemented in M_{llo}, these factors increase to $19\times$ for LUTs, $25\times$ for FFs and $12\times$ for power dissipation compared to a TCAM and $3.3\times/1.8\times/3.8\times$ when compared to StrideBV. The power dissipation for the TCAM was linearly extrapolated where necessary.

5.2.4 Summary on Hardware Centric Classification Circuits

The FPGA evaluation of different types of hardware circuits for network packet classification confirmed major differences with regard to resource usage, update latency, and power consumption. Given that in practical applications, a sufficient part of the rule set is static and not affected by rule set updates, a hybrid combination of different classification types can significantly reduce resource and power consumption, while still enabling dynamic updates of different sizes. This way, the circuit can be tailored for the specific application, which allows it to be implemented on smaller and cheaper FPGAs. A hybrid firewall like HyPaFilter+ could benefit from a hardware matching circuit with the capability for dynamic updates, as the slow software path could be avoided for simple rule updates.

5.3 Fast Hash Functions on FPGAs

Hash functions are used to calculate a fixed-size hash value from a given input of arbitrary length. They have numerous applications, e.g., hash tables, integrity protection, Bloom filters [14], or authentication, making them a vital component in almost any computer system. These applications are built on top of standard CPU-based systems as well as dedicated hardware like FPGAs. Nevertheless, the requirements for a fast and efficient algorithm differ substantially between software for CPUs and hardware description for FPGAs. The advantage of a hardware implementation lies in the potential for massive parallelization at a comparatively low clock rate. In practise, many fast hash functions used for hash tables were designed as CPU-optimized software algorithms and do not perform well when implemented in hardware [81].

This problem becomes even more relevant if the hash application requires multiple, independent hash values of the same key. Examples for these applications are hash tables with double hashing [15], cuckoo hashing [66], or Bloom filters [14]. When implementing this task in hardware, the developer has to choose between re-using one hashing instance, which increases the latency for the calculation, or implementing multiple hash instances at the cost of higher resource usage. As both methods have significant drawbacks, the question arises whether it is possible to exploit the fact that the required hash size is often significantly smaller than the actual size of the hash function's output. If there are no weaknesses in the output of a given hash function, the hash could simply be split into multiple sub-hashes. Although some authors argue that small flaws in the hash calculation are acceptable for hash tables when the full hash value is used [18], it is not clear if this is still the case when only parts of the hash are used.

Many non-cryptographic hash functions reveal issues when their *avalanche effect* is analysed [30, 115], in the sense that some input bits do not optimally propagate through the function. One of our goals is to determine the implications of those weaknesses with regard to our desired sub-hashes. It should be noted that a good avalanche effect of the function still does not necessarily imply there are no weaknesses in the hash, as can be seen for, e.g., the MurmurHash [118].

As previously mentioned, one must be aware that fast and efficient hash function designs for CPUs and hardware differ. Regarding the hardware implementation, the most important metrics of a hashing algorithm are resource utilization, latency l in clock cycles, and execution time as a result of the maximum possible clock rate. Hash algorithms usually rely on calculation rounds with feedback of the last round's result. Due to this constraint, typical hardware hash implementations

are not fully pipelined, meaning they are occupied until one calculation finishes. A significant fraction of common hash functions suffer from large latencies when implemented in hardware [81], making them less suitable for, e.g., high-speed, low-latency network applications. Furthermore, to gain full advantage of a highly parallelized processing pipeline, it is often necessary to process one input key per clock cycle. In order to do so, the hash functions must be implemented l times in hardware and used in a round-robin manner or implemented as a pipeline that can store the internal state for each of the l stages. When the algorithm is implemented as a dedicated processing block (e.g., a hard IP core), a higher clock frequency of the hashing function may mitigate the latency drawback. However, such cores are typically not common on standard systems and may further not be accessible for custom applications.

The growing importance of dedicated, feature-rich hardware components led to a shift in requirements when new standard algorithms are defined. For example, the winning candidate for the Secure Hash Algorithm 3 (SHA3) [104] was required to perform well in hardware. This raises the question whether such a hardware-optimized cryptographic hash function is more suitable even for non-cryptographic applications like hash tables than the non-cryptographic alternatives mentioned above.

This section evaluates properties of different hash functions for hardware implementation:

1. We show how statistical relevant weaknesses in the avalanche effect of a hash function can affect the uniformity of sub-hashes, as hardware implementations are more likely to be limited to simpler hash functions.

2. We examine the characteristics of several hash functions when implemented for a multi-hash FPGA use case.

3. Based on these results, we demonstrate that SHA3 is currently a better choice for many FPGA use cases regarding the trade-off between resource consumption and security in comparison to non-cryptographic hashes.

5.3.1 Hash Table Implementation

The key-value lookup accomplished by hash tables is important for a variety of networking tasks like stateful packet filtering, route lookup, or intrusion detection. Since dedicated hardware is increasingly used for these types of applications, hash table implementations for FPGAs have been widely discussed [18, 83].

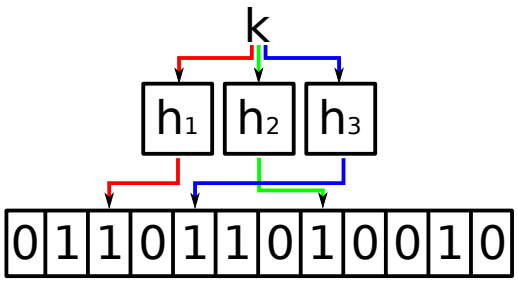


Fig. 5.13: Basic structure of a Bloom filter.

Bloom filters [14] can be a fast and efficient alternative when the only task is to query if a key is present in the filter. Since this is the case for many classification tasks in networking systems, Bloom filters are widely used in this field [17]. This type of filter is based on a bit array. For each key, k addresses to bit positions are derived by feeding the key to k hash functions, as depicted in Figure 5.13. An insertion sets the bits on each position to 1, while the lookup probes for a 1 on all positions. The lookup result can be a false positive, as the queried positions might have been set to 1 by the insertion of other keys. In contrast, false negatives are impossible. The variant *counting bloom filter* [65] utilizes a counter array instead and also allows the deletion of keys as long as the counters do not reach their maximum value. The feasibility of Bloom filters on FPGAs has been shown in [8, 83]. With memory lookups being the critical factor, SONG et al. suggested using Bloom filters to reduce the amount of hash table operations by first probing a Bloom filter if the lookup is required in the first place [83]. If the query is negative, no expensive hash table lookup is necessary.

Good hash functions are of major and ongoing interest [104, 81]. Countless hash functions—cryptographically secure or not—have been introduced, quite a few of which have been shown to have significant flaws with regard to the expected qualities of a good hash function [52, 118]. Hardware implementations of several hash functions were analysed in [81], with the result that most of them perform badly, causing a calculation latency too high for network processing applications [83].

When hash functions are used for hash tables, the main security concern arises from attackers being able to generate hash collisions with different keys. This can degrade the performance of hash tables and allow for denial-of-service (DoS) attacks [24]. Such flaws also led to security advisories, e.g., [117, 118]. BAR-YOSEF et al. were able to successfully attack the hash table in Linux' netfilter firewall [12], even though a randomization technique was implemented to protect from such attacks.

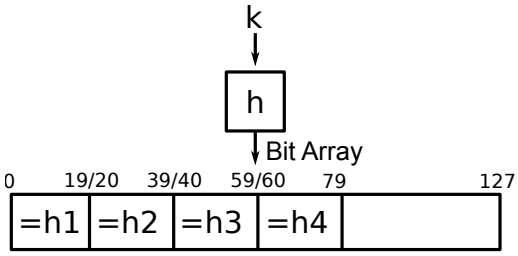


Fig. 5.14: Splitting one large hash value into independent sub-hashes for multi-hash applications.

From the variety of applications for hash functions, we focus on FPGA use cases requiring multiple, independent hash values for, e. g., hash tables using *open addressing* by double hashing [15], cuckoo hashing [66], or Bloom filters [14].

There are different ways of generating the required i independent hash values out of the same key:

1) using different hashing algorithms $h_i(k)$,

2) using the method of double hashing, where two different hash functions are employed to compute the hashes: $h_i(k) = (h_1(k) + i \cdot h_2(k))$,

3) mixing distinct seeds s_i to the key *before* feeding it to the same hash function $h_i(k) = h(k \oplus s_i)$, and

4) splitting one hash value into non-overlapping sub-hashes $h_i(k) = h(k)_i$, as depicted in Figure 5.14.

The drawback of the first two options is that they either lead to higher resource usage or increased latency, due to the fact that multiple hash functions need to be computed. For the third option, it can be chosen whether to implement multiple hash modules at the cost of logic resources or re-use one implementation and thereby increasing the latency until all results are calculated. The last option is the only option that saves both space and latency, but requires a hash function of sufficient quality and hash value length.

5.3.2 Attack Scenario

An attacker might try to generate *hash collisions* to degrade the performance for a DoS attack, since in case of collisions expensive computations must be performed [24]. If the hash function under consideration has statistical weaknesses

in one of its sub-hashes, such collisions are more likely to occur than for a hash with uniformly distributed outputs.

If a Bloom filter is used to reduce the hash table workload as suggested in [83], an attacker would want to provoke false positives by purposefully filling the filter with ones. In the worst case, each "real" query would then result in a false positive, thereby enforcing an expensive hash table lookup. Also in this case, the likelihood and degree of the attacker's success strongly depend on the uniformity of the hash function's output. It is helpful for the attacker if he can make assumptions on how changes in the input data affect the output data in another way than a pseudorandom behaviour, as it would be the case for a good hash function.

Non-cryptographic hash functions employed in practise are the Jenkins hash [46], as used in the Linux packet filter `netfilter` [25] or its successor SpookyHash [46]. Other examples include MurmurHash [118], CityHash [98], and SipHash [9]. The motivation for using these hashes are efficient implementations, and the mere requirement of uniform outputs. However, in applications like packet filters it is often not complicated to attack those kinds of functions if the attacker has control over their inputs [118]. In this section, we focus on a specific attack, where we exploit weaknesses in the so-called *avalanche effect* of the function [30] to actually generate hash values distributed in a non-uniform way, yielding a potential hash table DoS attack.

More formally, the avalanche probability for input bit i and output bit j of a hash function $h(x)$ is the probability $p_{i,j}$ that output bit j changes when input bit i is flipped, i.e.,

$$p_{i,j} = \frac{1}{2^n} \cdot |\{x \in \{0,1\}^n \mid h(x)_j \neq h(x^{(i)})_j\}|,$$

where $x^{(i)}$ is x with bit i flipped and n is the (in our case always finite) input size for the hash function h. Ideally, this probability should be close to $1/2$ for all admissible choices of i and j. This property can also be seen in the hamming distance of consecutive output values, which should be close to $\frac{\text{output size}}{2}$ for any flipped input bit i. However, the hamming distance alone is a necessary, but not a sufficient measure for the quality of the avalanche effect. To measure the distance to this desired probability, we define the *bias* to be $|p_{i,j} - 1/2|$. Intuitively, when this bias is large, the avalanche effect for the function differs for input bit i and output bit j considerably from optimal.

A good hash function should not have statistical weaknesses in this regard. In fact, there are many hash functions available which do not suffer from this weakness, most prominently cryptographic hashes such as SHA3 [28]. These functions have

the additional benefit that many other attacks (such as differential attacks) do not work either. A practical example of why even apparently small flaws can be problematic will be given in the next section.

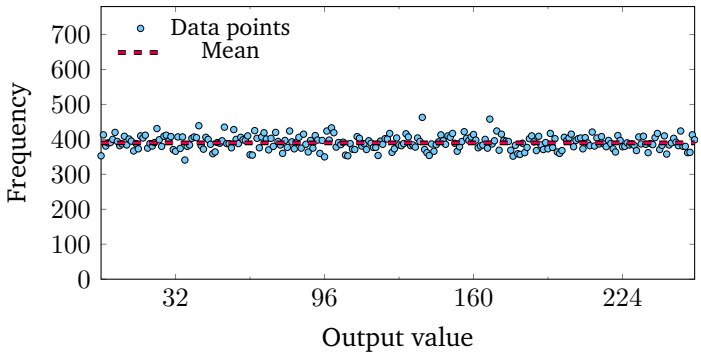
We begin our evaluation with an analysis of an avalanche-weak hash function and show how the described method of splitting a hash value into sub-hashes is affected. Afterwards, different examples of hashing algorithms are implemented for an FPGA and the results are evaluated.

5.3.3 Impact of Weaknesses in the Avalanche Effect

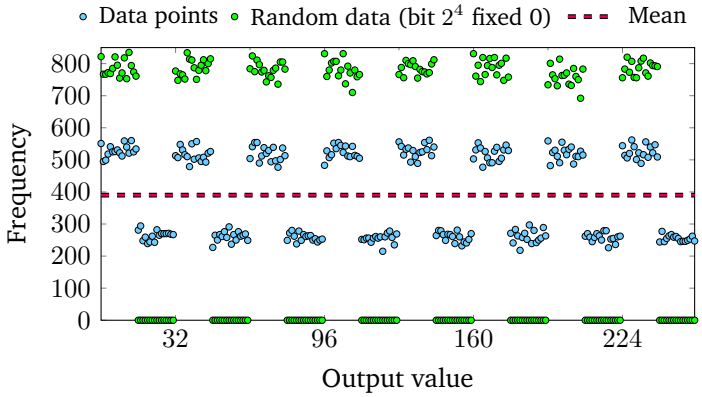
We first demonstrate that small weaknesses in the avalanche property of a hash function can be used to produce a non-uniform output distribution of the function, yielding potential hash table DoS attacks. For illustration of the effect, we purposefully selected the avalanche-weak Jenkins hash function and split the 32 bit hash value into four one-byte chunks. The other considered non-cryptographic hash functions, i.e., SpookyHash and SipHash, do not have any known weakness in this regard. The Jenkins function family comprises of different algorithms. We considered the latest `lookup2` and `lookup3` in our analysis.

For our experiments, we selected in a first step for both routines an input/output bit pair with large bias for 34 byte inputs—a flawless hash function would not allow finding such a pair. To find it, we empirically determined the bias for each input and each output bit by first selecting 10^5 random input values. We then successively flipped each input bit and counted the resulting flips over the 10^5 choices. We verified that this comparatively small number of samples is representative, since multiple runs of our experiments (with different random selections) gave comparable results. For further analysis, we selected a single input/output bit pair with bias larger than 0.025. Specifically, we considered for `lookup2` the pair $(i, j) = (248, 27)$ with bias 0.1685 and for `lookup3` the pair $(i, j) = (216, 25)$ with bias 0.03277.

In a second step, now with the fixed input/output pair (i, j), we generated 10^5 random inputs to the hash function, excluding those where the output bit j was 0. This was done in order to evaluate the effect of the bias, which would otherwise not be visible due to the random input data. We then flipped the bit i of the input and computed the distribution of the last two bytes of the function. Note that one of them contains output bit j. The results are depicted in Figure 5.15 and Figure 5.17. Figures 5.15b and 5.17b show a clear non-uniform distribution of the byte containing the output bit j. In Figure 5.15b, the origin of the pattern is emphasized by an additional set of generated random data where one bit has



(a) Non-critical 3rd byte.



(b) Avalanche-critical 4th byte.

Fig. 5.15: Distribution of Jenkins' `lookup2` sub-hashes scaled by 10^5.

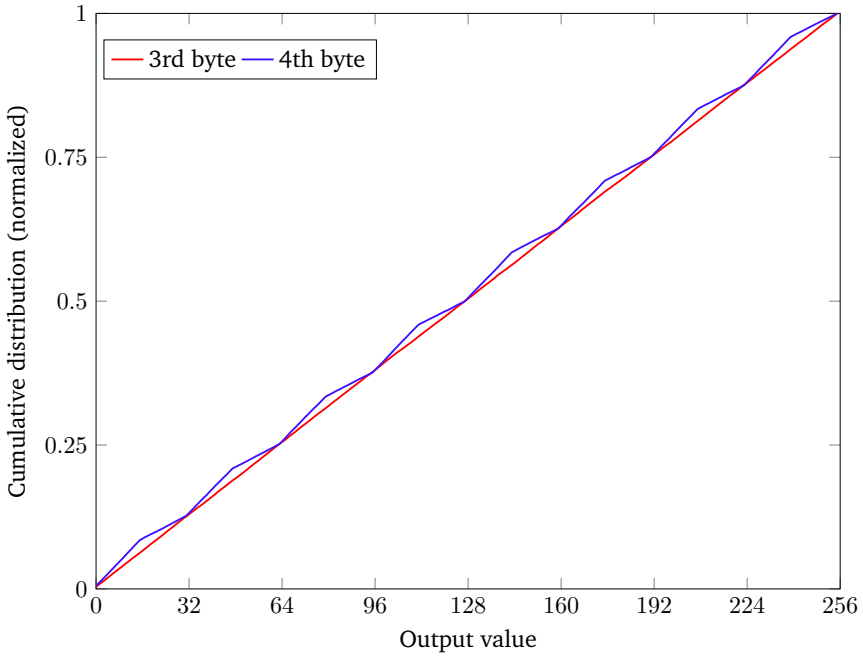
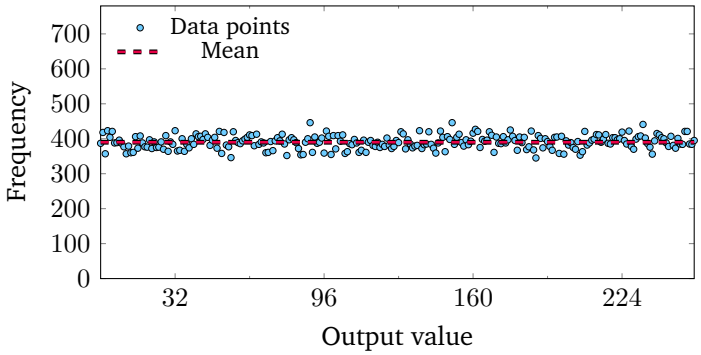
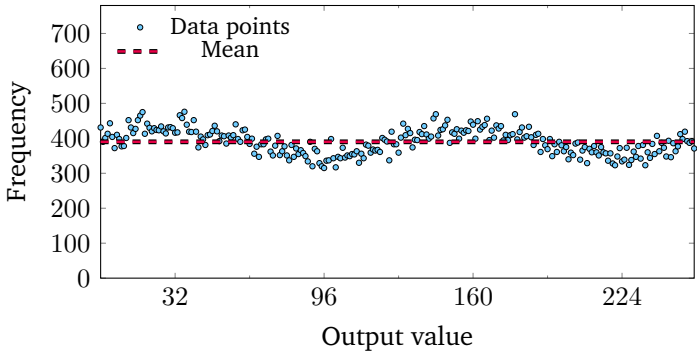


Fig. 5.16: Cumulative distribution of Jenkins' `lookup2` sub-hashes.



(a) Non-critical 3rd byte.



(b) Avalanche-critical 4th byte.

Fig. 5.17: Distribution of Jenkins' `lookup3` sub-hashes scaled by 10^5.

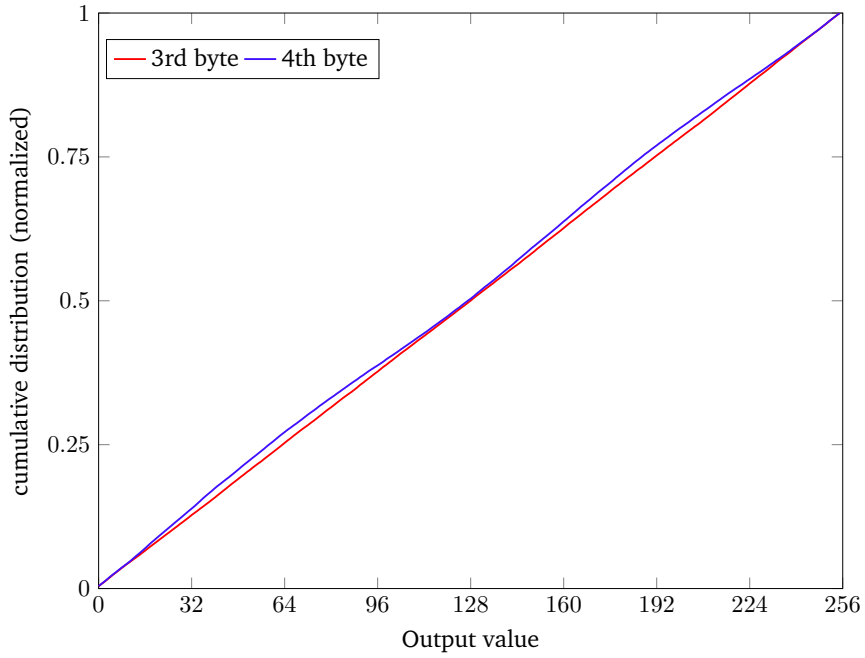


Fig. 5.18: Cumulative distribution of Jenkins' `lookup3` sub-hashes.

intentionally been set to zero. This bit in the 8 bit output data is the one with the weight 2^4, which derives from the selected bit pair ($j = 27$ in the 32 bit output data). The deviation can also be seen in the corresponding density plots in Figure 5.16 and Figure 5.18, where the observed data differs from the desired result of a straight line. For comparison, the byte not containing bit j is distributed close to the expected value $10^5/2^8 \approx 390$ for uniform distributions as can be seen in Figure 5.15a and 5.17a. This shows that even the presence of a single input/output bit pair with large bias can be used to easily induce skewness in the output distribution of the resulting hash.

To counteract this attack, we argue that one should be careful with the selection of a hash function and that even small statistical weaknesses can be exploited in practise. Nevertheless, we are aware that our experiments are rather a toy example than a fully-fledged attack on a concrete implementation.

5.3.4 FPGA Implementation Results

We selected four hash functions for implementation:

1) Jenkins (`lookup2`) [46], as it is used in Linux' netfilter firewall,

2) SpookyHash [46], the latest hash function of Bob Jenkins,

3) SipHash [9], the proposed alternative for hashes like MurmurHash and CityHash, and

4) SHA3 [28], as a current state-of-the-art cryptographic hash function.

While the avalanche-weak Jenkins was included in our evaluation for reference, CityHash and MurmurHash were not further considered due to reported weaknesses [98, 118].

All implementation results were determined for a fixed-size input key of 288 bit, which corresponds to the quadruple of two IPv6-addresses and two port numbers. The target frequency for the FPGA clock domain of all implementations was 200 MHz. The results were determined using Xilinx Vivado 2014.4 with a Virtex 7 690t, speed grade -2 as the targeted FPGA. Both Jenkins and SpookyHash were implemented natively based on the available source code [46]. For a comparison, we also applied high-level synthesis to directly derive a hardware design with Vivado HLS based on the publicly available C program code. The tool achieved comparable results to our native implementation. For SipHash, we

Hash	Size [bit]	LUTs	FFs	Lat. [CC/ns]	Ex. use case LUTs	Ex. use case FFs
Jenkins	64	2,874	3,419	76 / 380	436,848 (101.0%)	519,688 (56.0%)
SpookyHash	128	3,220	4,161	27 / 135	86,940 (20.1%)	112,347 (13.0%)
SipHash	32	944	789	52 / 260	196,352 (45.3%)	164,112 (19.0%)
SHA3-512	512	6,005	2,212	20 / 100	120,100 (27.7%)	44,240 (5.1%)

Tab. 5.2: Virtex 7 690t FPGA resource utilization.

used the referenced Verilog implementation [122], for SHA3 a SHA3-512 core from [109]. The latency was determined by simulating the HDL implementation of each core.

As can be seen in Table 5.2 for the evaluated hash functions and their hash value size, there are significant differences in terms of the usage of lookup tables LUTs, FFs, and the inherent latency (CC, in clock cycles and ns at 200 MHz) for the computation. To improve comparability, we included an example computation for a use case requiring eight independent 16-bit hash values of the same key, with the capability of processing one key per clock cycle. This means the hash core has to be replicated $n = \lceil \frac{\text{desired hash size}}{\text{native hash size}} \rceil \times$ latency times. Note that for SHA3, a smaller variant (e.g., SHA3-224) could be used since the hash size is larger than the required use case output size. The result for Jenkins is only given as a reference, as we already demonstrated a weakness with regard to the use case of splitting the hash value. The percentages illustrate clearly that a significant amount of the Virtex 7 FPGA resources are occupied for this use case. Moreover, a high latency alone can be a criterion for exclusion depending on the application. For comparison: in [83], the MD5 hash was deemed unsuitable for packet processing applications due to the latency of 64 clock cycles based on speed requirements present in the year 2005. Hence, from our evaluated candidates only SpookyHash and SHA3-512 can be considered suitable for low-latency FPGA applications.

As can be seen for the use case, the implementation results depend on the desired size and amount of independent hash values, as well as resource usage and latency of the hash functions. Two additional plots visualize this for different hash value sizes, provided that all can be split into independent sub-hashes of arbitrary size. Figure 5.19 assumes the total size is achieved by multiple, parallel hash modules. In contrast, Figure 5.20 shows how the latency is affected if the necessary calculations are executed in series on one single hash module, thus maintaining almost constant resource usage. Combined with the demonstrated, possible implications of non-cryptographic hash functions, we argue that nowadays the cryptographic hash SHA3 should be the default choice for hardware implementations, also for non-cryptographic applications.

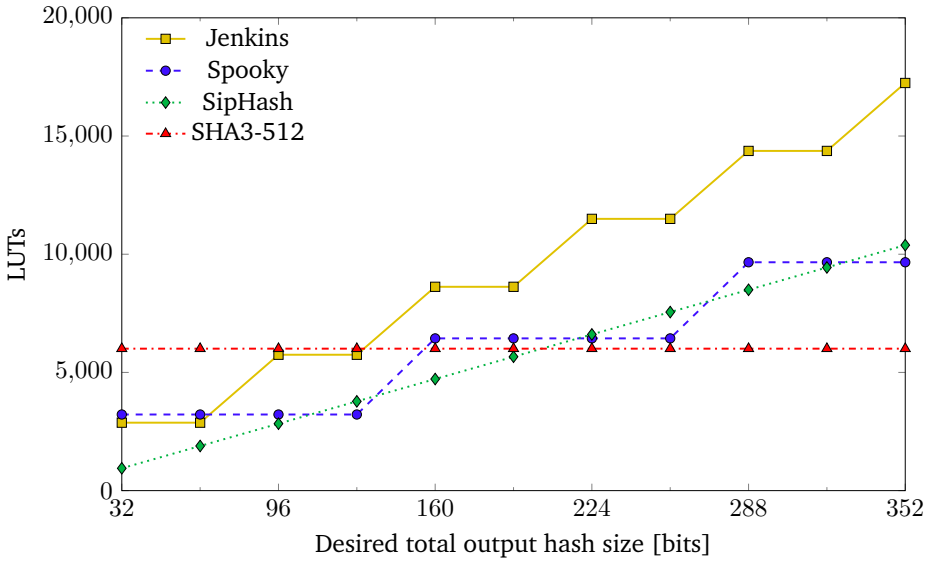


Fig. 5.19: FPGA utilization for parallel hash cores: LUT usage.

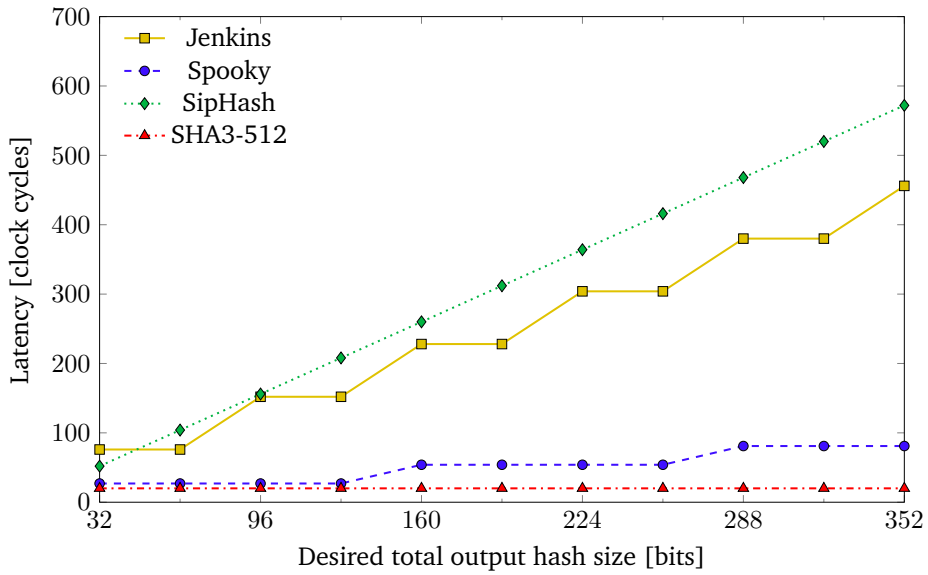


Fig. 5.20: Latency for serial hash core usage.

5.3.5 Summary of Hash Functions on FPGAs

Good hash functions are essential for a variety of applications, with hash-based data structures being just a few of them. For this evaluation, we focused on use cases like certain types of hash tables and Bloom filters, where several independent hash values of the same key are required. We demonstrated that the method of splitting one hash value into sub-hashes for multi-hash use cases may show non-optimal behaviour if the used hash function suffers from weaknesses with regard to the avalanche effect. Further, we analysed the recent cryptographic hash SHA3 and compared it against common hash table hash functions as an alternative for hardware applications. Our results show that most hardware hash applications benefit from the use of SHA3 instead of non-cryptographic hashes optimized for CPU-based systems.

5.4 Summary

Network packet processing involves a large number of different tasks on different levels. In order to fulfill the demand for higher throughput, there is an increasing interest to migrate those tasks to faster hardware components. However, more than with standard software implementations, system designers have to cope with limited hardware resources and capabilities. In this chapter, we demonstrated how a hybrid conjunction of logic-level optimized and dynamic packet classification circuits can be used for a resource-efficient realization. While the LLO circuit implements policies using a significantly smaller resource footprint than a generic circuit, a smaller variant of the latter is used to preserve the dynamic update capability. Given the property that in many setups, a fair share of the policies is static, this approach allows to implement the required design on smaller and cheaper FPGAs.

One further crucial sub-task for many network processing applications is the calculation of hash values. A comprehensive analysis revealed that popular, CPU-optimized hash algorithms for this type of applications are not suitable for FPGA implementation. In particular, no advantage could be gained from parallelization, while the lower clock rate of an FPGA compared to modern CPU results in even higher latencies. Further, many of these highly CPU-optimized hash functions suffer from weaknesses with regard to security properties. This prohibits, e.g., the hash value to be split into independent parts, as the effect of the weaknesses may increase. We compared the modern, cryptographic hash function SHA3 with classical hash table hash functions targeting a scenario where multiple sub-hashes are required. We could demonstrate that, although SHA3 exceeds the requirements, it outperforms these functions due to the hardware-centric specification.

FPGA-based network packet processing has the potential for adapting to further tasks and further optimizations. Nevertheless, in all work introduced in Chapters 3, 4 and 5, the FPGA part required a specific implementation and the utilization of FPGA-based hardware. We will therefore further examine how standard networking hardware like SDN devices can be used to apply the hybrid approach.

6 Hybrid Approach Applied to SDN

This chapter is based on previous work by the author [4]. The general setup and experiments were the result of a joint effort and should also be attributed to fellow co-authors.

6.1 Overview

For larger scale networks, SDN has become a widely adopted concept [53]. The separation into control plane, data plane, and SDN applications allows administrators to flexibly adapt the network to new requirements, without the need to replace hardware components. In this chapter, we use such SDN hardware as a replacement for the FPGA-based packet classification system, as SDN devices are available COTS and are not only cheaper, but also often already available and integrated in existing network infrastructure. On the downside, the restriction to standardized features of SDN hardware will prevent using some of the specific features we achieved by our own implementation of the hardware classification circuit. In particular, there is no dedicated communication channel available for the forwarding of matching information as it is used in Chapter 3.

Typical SDN hardware is well suited for simple classification tasks. An example is simple stateless classification based on IP addresses or port numbers. Its capabilities are similar to hardware classification systems as introduced in Section 5.2—in fact, they are often based on TCAMs. As network security is a topic of major interest, there are also approaches to implement firewalls based on SDN natively [105, 41]. As introduced in Chapter 2, firewalls often rely on advanced matching criteria like *deep packet inspection* (DPI) [26]. A typical dedicated SDN switch hardware, however, cannot be utilized for these complex tasks.

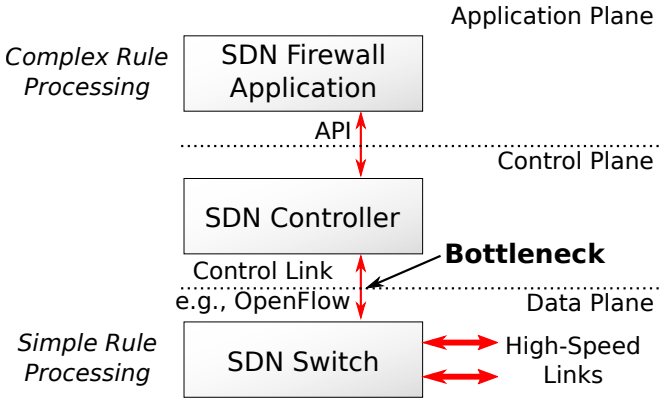


Fig. 6.1: Basic setup for an SDN firewall. Here, rules with complex checks force packets via the bottleneck control link to the SDN application, which executes the complex checks.

```
-d 192.0.2.1/32 -p tcp --dport 80 -j ACCEPT                          # Rule R1
#-------------------------------------------------------------------------------
-d 192.0.2.3/32 -p udp --dport 8200 -m string                        # Rule R2
   --string "SELECT"         --algo bm -j DROP                       #
#-------------------------------------------------------------------------------
-d 192.0.2.7/32 -p udp --dport 53 -m string                          # Rule R3
   --hex-string "|11|2|00|" --algo bm -j DROP                        #
#-------------------------------------------------------------------------------
-d 192.0.2.9/32 -p udp --dport 53 -j ACCEPT                          # Rule R4
```

Listing 6.1: Example rule set $\mathcal{R}$ in `iptables` syntax.

Therefore, complex analysis cannot be handled solely on the fast data plane, but involves the application plane via the control plane in order to enable the SDN firewall application to make a classification decision. Nevertheless, the link between the switch and the controller is mainly designed for relatively infrequent control messages and not intended to be used for large amounts of data. Hence, it represents a bottleneck when many packets need to be sent to the controller [27]. An example setup of a native SDN firewall is sketched in Figure 6.1.

For demonstration, an example rule set is given in Listing 6.1. Rules R_1 and R_4 can easily be implemented by an SDN switch in the data plane, since both only specify simple header field checks. On the contrary, R_2 and R_3 rely on string matching (payload analysis), which exceeds the capabilities of typical SDN hardware.

Due to the bottleneck to the application plane, the performance of such a firewall greatly depends on the range of operations the switch can handle directly at the data plane, which is dependent on the switch hardware and the protocol the SDN switch supports. There is currently no development towards SDN switches that would natively implement features comparable with standard software firewalls [64]. Nevertheless, the high performance classification capabilities of SDN hardware motivates their integration into a hybrid concept that employs a direct policy separation as introduced in Sections 3 and 4 and sketched in Figure 6.2.

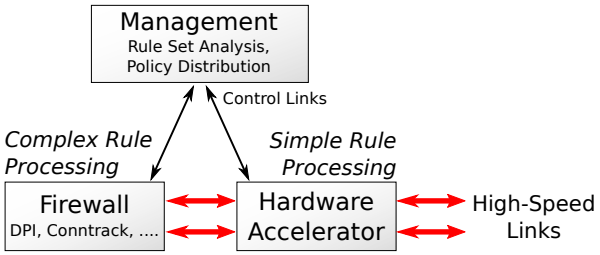


Fig. 6.2: Basic setup of a hybrid firewall with policy separation for differing capabilities of the compartments. Traffic is kept on fast links, while control links have modest requirements.

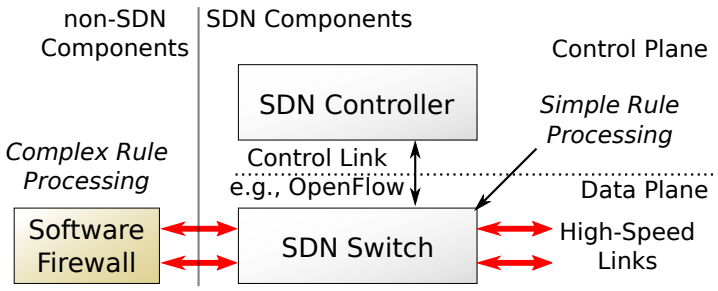


Fig. 6.3: FIREFLOW setup for an SDN firewall. FIREFLOW offloads the complex rule processing to a software firewall and thus removes the bottleneck via the control link.

The proposed FIREFLOW approach also uses a central rule management tool for analysing and distributing the firewall's rule set to a standard software firewall and an SDN switch via the OpenFlow protocol. SDN components are easily available at reasonable cost and provide high link capacities. By using SDN hardware, FIREFLOW can seamlessly be integrated into existing infrastructure. The FIREFLOW approach handles simple decisions within the data plane's scope of the SDN switch with full line speed, as sketched in Figure 6.3. From another point of view, the SDN switch can be seen as a high-performance shunting mechanism for a standard software firewall. Note that the FIREFLOW architecture allows the use of a P4-enabled switch or network card as well [16]. Nevertheless, in this chapter we focus on the integration of existing OpenFlow infrastructure whereas leveraging P4 remains future work.

As described in Chapters 3 and 4, one key challenge of the hybrid design is the distribution of the policies and handling rule set dependencies. To this end, a similar strategy based on HSA is applied to reduce the workload in FIREFLOW. In terms of possible performance gains, we further rely on properties of rule sets and traffic that have been shown in these sections. In particular, rule sets must not have too many dependencies with regard to overlap in header space; whereas the majority of the traffic must be processable by using simple policies only (also shown in [7]).

In summary, this chapter will answer the third research question (application of the approach to other packet classification systems) and provide the following contributions:

- We describe the challenges and present solutions for hybrid setups of SDN and standard software firewalls.

- We show how the approach can be applied as a performance-increasing measure for software firewalls.

- We demonstrate how existing SDN infrastructure can be upgraded seamlessly into a fully fledged firewall.

- We evaluate the feasibility of FIREFLOW and demonstrate an over 23-fold classification speed-up with real-world rule sets compared to a software firewall.

The remainder of this chapter is structured as follows: in Section 6.2, we discuss relevant related work. Afterwards, we describe the setup of a FIREFLOW system in detail in Section 6.3. The following Section 6.4 is dedicated to explain the underlying packet routing algorithm and interaction in FIREFLOW. An evaluation by measurement in Section 6.5 shows the effectiveness of the approach and other relevant parameters. Finally, we summarize our results in Section 6.6.

6.2 Related Work

Previous work in the field focuses on three major topics:

1) Proactive SDN firewalls,

2) Reactive SDN firewalls, and

3) Hybrid SDN software firewalls.

Proactive SDN firewalls implement security policies by statically deploying rules in the SDN switches. This is the predominant mode of implementation in real world controllers (e.g., [105, 125]), as it is easy to implement, resilient, and efficient. However, the size of the rule memory in SDN switches is fixed and limits the deployment of large policies.

In the literature, *reactive* SDN firewalls are more prominent (e.g., [23, 41, 47, 67, 95]). In general, security policies are kept in the controller and a new flow is checked against them. If accepted, suitable flow rules are pushed into the switches. None of these approaches are able to deal with complex policies including rules that are not implementable by the means offered by the switch hardware. Without further research, the naive approach to enable complex policies through reactive SDN firewalls would include the handling of the corresponding flows in the controller abandoning further rule establishment.

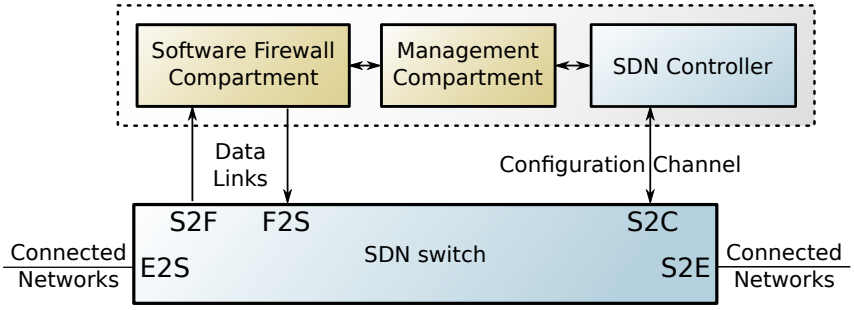


Fig. 6.4: High-level setup of FIREFLOW.

Hybrid designs focus on offloading network traffic by combining SDN switches as accelerators and classical firewall appliances. In [62], a Linux `netfilter` based firewall is used to filter traffic and to offload predefined connections, e.g., data transfers known to be resource intensive, in a stateful manner automatically. A similar approach is performed by [40] which supervises a legacy firewall with sFlow and offloads congested connections. A major drawback of these two approaches is the limited shared space available for flows and policies in the SDN switch. One rule can create a large number of flows, which means storing single flows instead of policies is also less resource efficient. Thus, the feasibility of this approach highly depends on the capacity of the switches. Moreover, offloading a flow prevents further, deeper analysis of its contents.

In [60], a hybrid design featuring SDN and NFV is proposed. A similar approach is demonstrated in [68]. The design aims to support complex filtering through virtualised legacy firewalls while leveraging the SDN to offload suitable connections into the fast switching hardware. However, a performance study is still missing.

The drawbacks of using the control plane for actual traffic in terms of latency have been evaluated in [27]. The authors showed that a significant delay is caused by packets' context switches, while the delay through the processing itself is present, but short in comparison. FIREFLOW circumvents this bottleneck by keeping all data traffic within the context of the data plane.

6.3 System Setup

A FIREFLOW system is intended to be either built on top of a standard SDN infrastructure or as a measure of accelerating a standard software firewall. As depicted in Figure 6.4, several logical components are utilized.

SDN Switch The SDN switch provides the data plane switching functionality. It is connected to the software firewall via high-speed network links, and the SDN Controller via a control link.

SDN Controller The SDN controller is the central element of the control plane. It uses the OpenFlow protocol [61] to communicate with the SDN switch and can be configured from the management compartment.

Software Firewall A standard software firewall like `iptables` [127], `pf` [119], or `ipfw` [111] can be used as the back-end for FIREFLOW. As the traffic is not altered or encapsulated, no special modifications to the firewall are required. This also enables the use of commercial firewall appliances. However, it needs to be remotely administrable via a suitable protocol, as it is not statically configured by a local administrator.

Management Compartment The management compartment keeps track of the configured rule set and is used to administer the FIREFLOW system. It is the only component that needs to be exposed to the administrator. The management tool has both access to the SDN controller to set rules on the SDN switch and the software firewall in order to configure the rule set.

For the remainder of this chapter, we define the following naming convention for the SDN switch network ports as shown in Figure 6.4:

1) SDN switch to and from external networks *E2S* (ingress) and *S2E* (egress),

2) SDN switch to and from software firewall *S2F* and *F2S*,

3) SDN switch to SDN controller (both directions) *S2C*.

The SDN controller, the software firewall, and the management tool can be placed onto one single physical system, or separated into different virtual or physical machines. The connections between SDN switch and software firewall should be of sufficient capacity to deal with a reasonable fraction of the data plane traffic. Depending on the scenario, however, they may be of significantly lower bandwidth than the external connections from and to the SDN switch (see the evaluation results in Section 6.5).

There are no specific requirements for the link between SDN switch and SDN controller, which is typically a low-performance connection [27, 53]. Configuration updates from the management tool are distributed either directly within

```
-s 192.0.3.0/24 -p tcp --dport 80 -j ACCEPT                    # R_A1
#-------------------------------------------------------------------------
-s 192.0.4.0/24 -p udp --dport 53 -j ACCEPT                    # R_A2
```

Listing 6.2: Example rule set $\mathcal{R}_\mathcal{A}$.

```
-s 192.0.3.0/24 -m string --string "SELECT" --algo bm -j DROP  # R_B1
#-------------------------------------------------------------------------
-s 192.0.1.0/24 -m string --string "BAD" --algo bm -j DROP     # R_B2
#-------------------------------------------------------------------------
-s 192.0.2.0/24 -p tcp --dport 80 -j ACCEPT                    # R_B3
#-------------------------------------------------------------------------
-s 192.0.4.0/24 -p udp --dport 53 -j ACCEPT                    # R_B4
```

Listing 6.3: Rule set $\mathcal{R}_\mathcal{B}$.

the operating system's context if placed on the same machine or via any suitable communication channel, e. g., using *Secure Shell* (SSH).

6.4 Hybrid SDN Packet Processing

The key for achieving significant performance gains with FIREFLOW lies in the packet routing decision and fast software processing. Hence, our goals are to keep as much workload as possible on the fast SDN data plane and reducing the software firewall policies to the necessary minimum. A further goal is limiting the size of the rule set on the SDN switch: while typical hardware classification approaches, such as TCAMs, achieve line-speed classification rates, the size of the configuration memories used by these techniques is strictly limited [31, 39, 48, 49]. For example, in our evaluation setup, the Aruba 3810m switch [108] could hold less than $1\,500$ rules with the desired expressiveness. Rule set sizes exceeding the limit will therefore be forced to be processed in significantly slower compartments, in the worst case the SDN controller itself.

To demonstrate the operation of FIREFLOW, we start with the example rule set $\mathcal{R}_\mathcal{A}$ shown in Listing 6.2. Note that for better readability, the chain parameter `-A FORWARD` used in `iptables` is omitted in all following examples. Here, only IP addresses, port numbers, and protocol types are examined. As both rules only consist of simple checks, they can be implemented easily as SDN hardware-only rules on the SDN switch that forwards matching packets from port E2S to port S2E.

The next example in Listing 6.3 introduces two rules R_{B1} and R_{B2} for string matching. As SDN switches are usually not capable of payload analysis, FIRE-

```
-s 192.0.3.0/24 -m string --string "SELECT" --algo bm -j DROP          # R_C1
#-------------------------------------------------------------------------------
-s 192.0.2.0/24 -m string --string "BAD" --algo bm -j DROP             # R_C2
#-------------------------------------------------------------------------------
-s 192.0.2.0/24 -p tcp --dport 80 -j ACCEPT                            # R_C3
#-------------------------------------------------------------------------------
-s 192.0.4.0/24 -p udp --dport 53 -j ACCEPT                            # R_C4
```

Listing 6.4: Rule set $\mathcal{R}_C$.

FLOW will therefore redirect correspondent packets to the software firewall, which contains the rules R_{B1} and R_{B2}.

The redirection can be accomplished as follows. A first method, called *rule divert*, places two correspondent SDN hardware rules into the switch before rules R_{B3} and R_{B4}. This is given by the function PARTITION_RULE_SET_RULE_DIVERT in Algorithm 5, which is based on [5] and takes $\mathcal{O}(n^2)$ time. These additional rules correspond to the geometric reduction of the complex rules (i.e., R^-_{B1} and R^-_{B2}) and divert all traffic matching the addresses and protocol from switch port E2S to port S2F. Hence, the switch will classify the packet against a part of each complex rule that it can handle in hardware, leaving the rest to the software firewall. Further, this method takes care of the dependencies inherently.

Two further SDN hardware rules will forward any traffic from the software firewall to the corresponding switch output ports (F2S to S2E). This action will be applied by a generic rule at the bottom of the rule set. In order to respect all rule dependencies for both methods, an HSA can be used to identify all simple rules that interfere with any of the complex rules not present in the SDN switch. The software firewall must then be loaded with either the full rule set, or only complex and HSA-dependent rules. The latter is comparable to the *HSA cut set strategy* described in Section 4.5.1. It should be noticed that with first rule match processing order, there is no further interference between the shown rules.

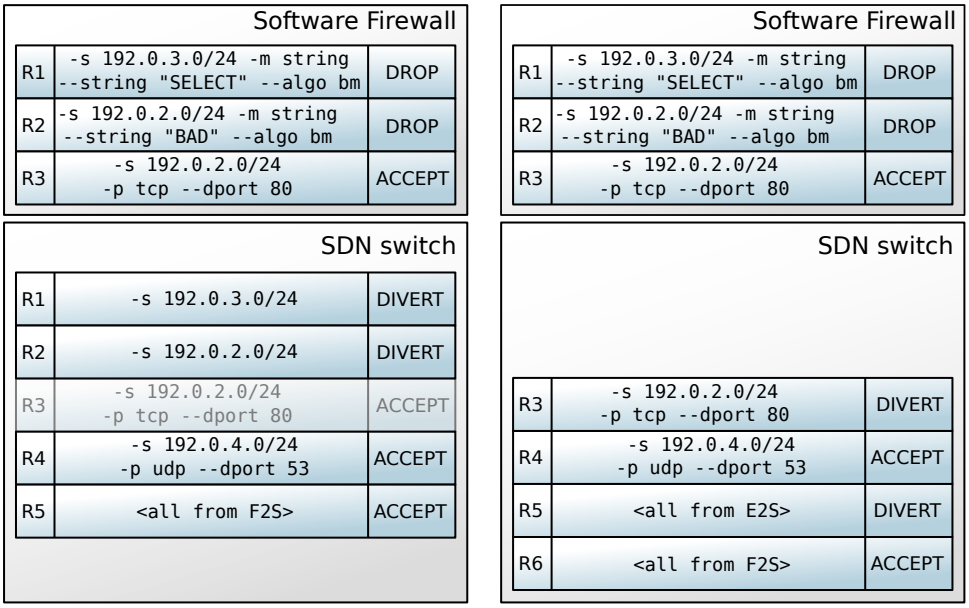
The next example shown in Listing 6.4 also includes string matching rules similar to the former example. Like in the example of Listing 6.3, FIREFLOW would place diverting rules to the software firewall for packets matching the stated subnets and software rules including the string matching part in the software firewall. However, in contrast to the former example, this rule set contains a possible interference between rules R_{C2} and R_{C3}. This can occur whenever a packet from the source network 192.0.2.0/24 arrives, but does not match the string in R_{C2} in the software firewall. It can still match rule R_{C3}, though. Therefore, one must take care of these dependencies and place appropriate rules in the software firewall as well. In this example, the software firewall would contain the rules R_{C1}, R_{C2}, and R_{C3}.

Algorithm 5 Partition rule set $\mathcal{R}$.

```
1:  function OVERLAP(Rule R_i, Rule R_j)
2:      return G(R_i^-) ∩ G(R_j^-) ≠ ∅

3:  function IS_COMPLEX(Rule R_i)
4:      return G(R_i^-) ≠ G(R_i)

5:  function IS_HSA_DEPENDENT(Rule R_i, Rule set R_complex)
6:      for Rule R_j ∈ R_complex do
7:          if OVERLAP(R_i, R_j) then
8:              return true
9:      return false

10: function MAKE_DIVERT_RULE(Rule R_i)
11:     R_div ← COPY(R_i)
12:     set R_div's action to divert
13:     return R_div

14: function PARTITION_RULE_SET_RULE_DIVERT(Rule set R)
15:     R_simple ← [ ]
16:     R_complex ← [ ]
17:     for Rule R_i ∈ R do
18:         if IS_COMPLEX(R_i) then
19:             R_complex ← R_complex + [R_i]
20:             R_simple ← R_simple + [MAKE_DIVERT_RULE(R_i^-)]
21:         else if IS_HSA_DEPENDENT(R_i, R_complex) then
22:             R_complex ← R_complex + [R_i]
23:             R_simple ← R_simple + [R_i]
24:         else
25:             R_simple ← R_simple + [R_i]
26:     return (R_simple, R_complex)

27: function PARTITION_RULE_SET_DEFAULT_DIVERT(Rule set R)
28:     R_simple ← [ ]
29:     R_complex ← [ ]
30:     for Rule R_i ∈ R do
31:         if IS_COMPLEX(R_i) then
32:             R_complex ← R_complex + [R_i]
33:         else if IS_HSA_DEPENDENT(R_i, R_complex) then
34:             R_complex ← R_complex + [R_i]
35:             R_simple ← R_simple + [MAKE_DIVERT_RULE(R_i)]
36:         else
37:             R_simple ← R_simple + [R_i]
38:     return (R_simple, R_complex)
```

A second method, called *default divert*, can be used if it is desirable to reduce the number of SDN hardware rules. It is not necessary to insert the geometric reduction R^- of all complex rules into the SDN switch. Instead, only the subset of all simple rules can be used, with the HSA-dependent rules set to divert. All other divert operations are handled by a default divert rule at the bottom of the switch rule set. This method is given by the function PARTITION_RULE_SET_DEFAULT_DIVERT in Algorithm 5, again with time complexity $\mathcal{O}(n^2)$.

Figures 6.5a and 6.5b show a comparison of both diversion methods for the given rule set. It should be noted that in Figure 6.5a, rule R_3 in the SDN switch can never be hit due to the complete overlap of the diverting SDN switch rule R_2.



(a) Rule divert. **(b)** Default divert.

Fig. 6.5: Packet diversion methods for example rule set $\mathcal{R}_\mathcal{C}$ of Listing 6.4 in comparison.

```
-s 192.0.3.0/24 -m string --string "SELECT"                                    # R_D1
 --algo bm -j DROP                                                             #
#-------------------------------------------------------------------------------------
-s 192.0.3.0/24 -p tcp --dport 21 -j ACCEPT                                    # R_D2
#-------------------------------------------------------------------------------------
-p tcp --dport 80 -j DROP                                                      # R_D3
```

Listing 6.5: Rule set $\mathcal{R}_\mathcal{D}$.

We call such rules *dead rules*. A similar case can be seen in Figure 6.5b, where rule R_3 is redundant, as all matching packets would also be diverted by rule R_5. However, this is not necessarily applicable for all simple rules in all rule sets. In particular, take the rule set example in Listing 6.5. When looking at the matching criteria, one can see a partial overlap between the complex rule R_{D1} and the simple rule R_{D2} (source address), between simple rules R_{D2} and R_{D3} (protocol), but not between R_{D1} and R_{D3}. If R_{D2} would be omitted due to the overlap with R_{D1}, rule R_{D3} could be hit by packets that would otherwise match R_{D2} before, leading to an incorrect classification result. For this reason, the HSA dependency analysis is necessary. Although the detection and removal of dead rules is not implemented in the setup introduced in this chapter, it can readily be integrated into the partition algorithms by, e. g., applying the redundancy removal technique presented in [55].

Figures 6.6 and 6.7 illustrate the former example of the rule set in Listing 6.4 (sketched in Figure 6.5a) with two different packets, using the rule divert method. Both packets are diverted to the software firewall due to the required string match-

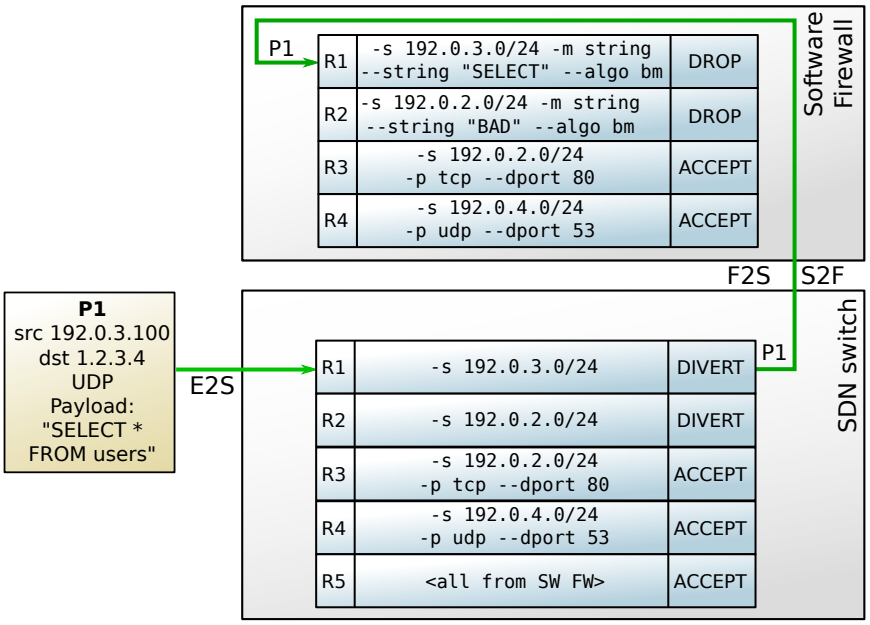


Fig. 6.6: Packet dropped due to complex software firewall rule (rule divert method).

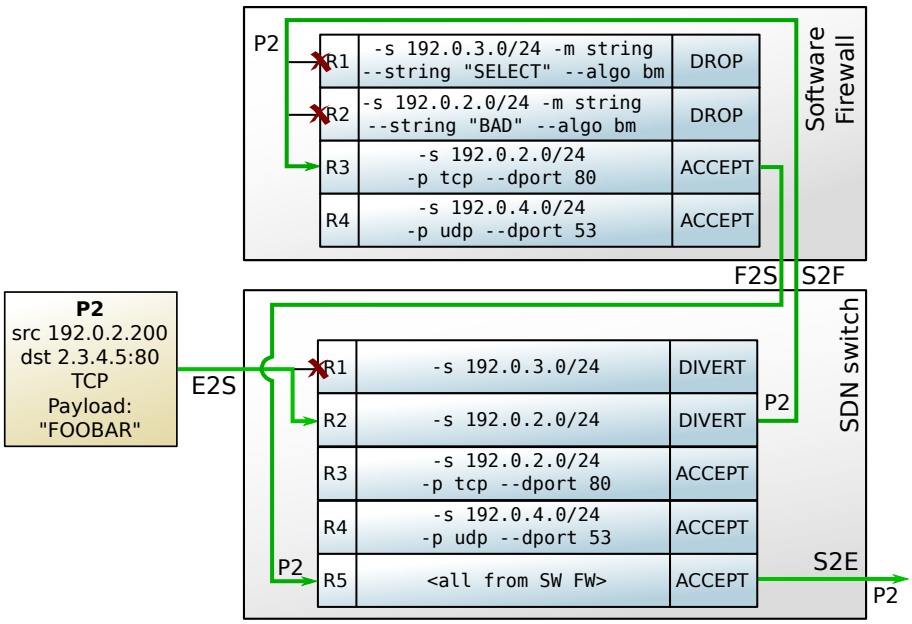


Fig. 6.7: Packet accepted due to complex software firewall rule (rule divert method).

ing operation. Packet P_1 matches the corresponding DROP rule and is therefore discarded directly at the software firewall. Packet P_2, however, does not match the string part in software rule R_2 (which corresponds to R_{C2}) and must be further analysed by the following rules. It matches software rule R_3 (corresponding to R_{C3}), redirecting it back to the SDN switch for further transmission by SDN rule R_5. Due to the approach of partitioning the rule set directly, no packet reordering takes place for any flow during normal operation.

6.5 Evaluation

In this section, we evaluate the packet classification performance gain that can be reached with FIREFLOW under different conditions. The performance of the hybrid setup is assessed in relation to a standard software firewall. As in Chapters 3 and 4, a set of synthetic and real-world firewall rule sets provides the basis for our measurements. We compare overall packet classification rates and packet loss due to complex rule processing. We further measure the time required for deploying rule set updates of different size.

6.5.1 Evaluation Setup

The test setup is similar to the one used for evaluating HyPaFilter and HyPaFilter+ in Chapters 3 and 4. Again, the classification performance is the main evaluation goal, so we aim at placing a high workload on the classification engine. Therefore, the payload is kept small and packets are sent at the highest rate possible. As shown in Figure 6.8, a setup of two machines is used to generate and count packets (called sender and receiver). These machines are equipped with an Intel E3-1270 CPU and a dual-port 10 Gbit/s NIC, which is also used for the connection to the SDN switch.

A bridging-type firewall setup is realized by placing the SDN switch in between. An additional system is used for the SDN controller, the software firewall, and the FIREFLOW management tool. This machine uses an Intel Xeon E3-1230 CPU, 16 GB RAM and is connected to the SDN switch via 10 Gbit/s links. Version details can be found in Table 6.1.

We generated ten synthetic rule sets of 1 100 rules each with the tool ClassBench [89]. These rules contain checks on IP address ranges, port ranges and protocol type. The number of rules was chosen to leave enough distance to the maximum number of rules on the switch, which, when exceeded, results in the automatic incorporation of a slow software-based compartment in the switch. In

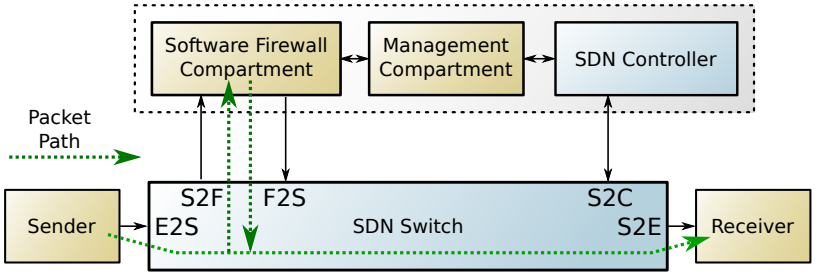


Fig. 6.8: Evaluation setup.

SDN Controller	Ryu 3.3
SDN Switch	Aruba 3810m [108]
SDN Protocol	OpenFlow 1.0
Software Firewall	`iptables` v1.4.21

Tab. 6.1: Version details.

addition to the synthetic rule sets, the three real world rule sets that we already used for evaluating HyPaFilter and HyPaFilter+, allowed us to compare the synthetic results to measurements closer to real conditions. In preparation, all rules were set as UDP rules with the action ACCEPT. This was done in order to focus on the classification performance. The Aruba 3810m switch supports a total number of $10\,000$ entries in the IPv4 hardware-based routing table [108]. However, with the desired expressiveness that also requires matching port numbers and protocol types, this number drops to less than $1\,500$ entries—the exact number depends on the rule set's characteristics.

In Section 6.4, we described two different diverting methods for recombining traffic of the software firewall. Throughout this evaluation, the rule divert method was used. A comparison measurement using the default divert method with this setup showed no notable difference with regard to the classification performance, which was expected due to the $\mathcal{O}(1)$ classification in the switch hardware. Thus, this step takes the same time for both approaches. Nevertheless, the default divert method is more resource efficient concerning the switching table, since it thins out the deployed simple rule set.

Traffic generation relies on trace files, which were generated by ClassBench's *trace_generator* according to the rule sets. The sender generates packets based on the trace files, which will hit the rule set evenly distributed.

6.5.2 Reference Measurement

In order to measure reference performance values, the same firewall tasks that were used to evaluate FIREFLOW are applied to the software firewall as stand-alone setup. The test rule sets are first configured to contain only simple rules. Then, we adapt an increasing number of rules to require a complex string matching operation. For the reference measurement, this only increases the processing complexity for the software firewall. In the following hybrid setup evaluation, however, these rules force a diversion of packets from the hardware accelerator to the software firewall. We also evaluated a native SDN firewall setup using the SDN controller via OpenFlow for complex rule processing, with the expected constraints due to using the control link for data traffic. For each of the rule sets, $k \in \{50, 100, 150, \ldots, 1\,100\}$ rules at equally spaced positions are modified for complex processing and the setup is updated via the management tool. Then, for each k, ten test cycles of 20 seconds are conducted, counting the number of successfully classified and forwarded packets. The number of packets arriving at the ingress port of either setup was about 30 million in 20 seconds.

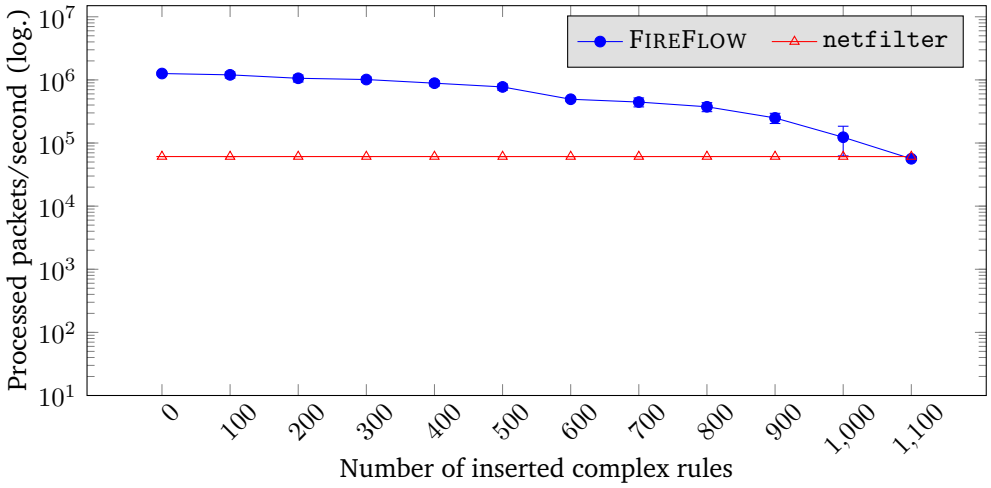
6.5.3 FIREFLOW as a Software Firewall Accelerator

The evaluation of the hybrid system follows the same procedure as the reference measurement. The software workload for simple rules is expected to be reduced and instead of using the SDN control channel, complex decisions are diverted to the software firewall compartment.

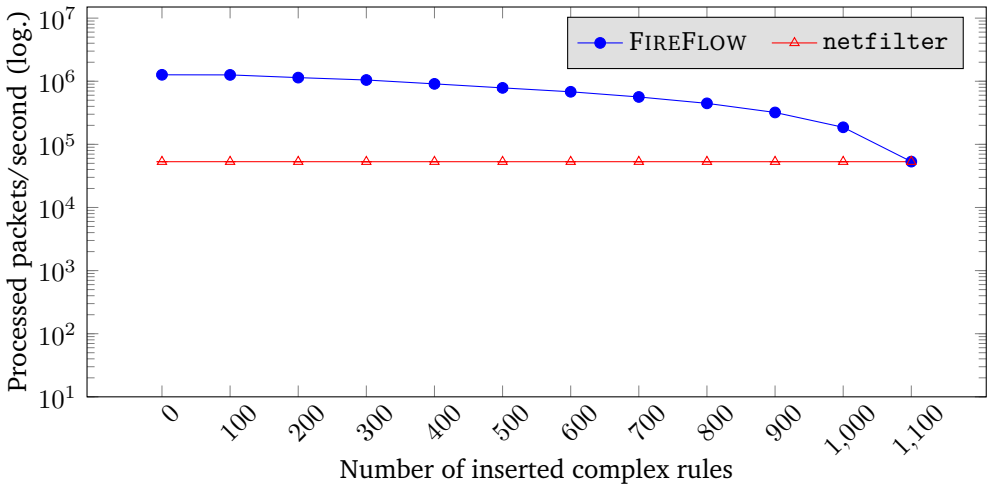
In Figure 6.9, one can clearly see an almost constant number of packets processed in the reference measurement, where all packets are solely classified by the `netfilter` software firewall. The increasing number of complex rules only has a minor effect on these numbers. This shows that the limitation of the software firewall arises from the software processing overhead rather than from the complexity of the actual rules. Figures 6.9a and 6.9b show the performance gain of FIREFLOW when compared to the reference software firewall. FIREFLOW can improve the performance especially when many simple rules can be offloaded to the SDN switch, where it reaches a 23-fold performance gain.

6.5.4 FIREFLOW as an SDN extension

In contrast to the software firewall reference, complex rules in a plain SDN firewall setup force packets to be sent to the SDN controller. Note that while in the case of FIREFLOW, the software firewall actually applies the complex

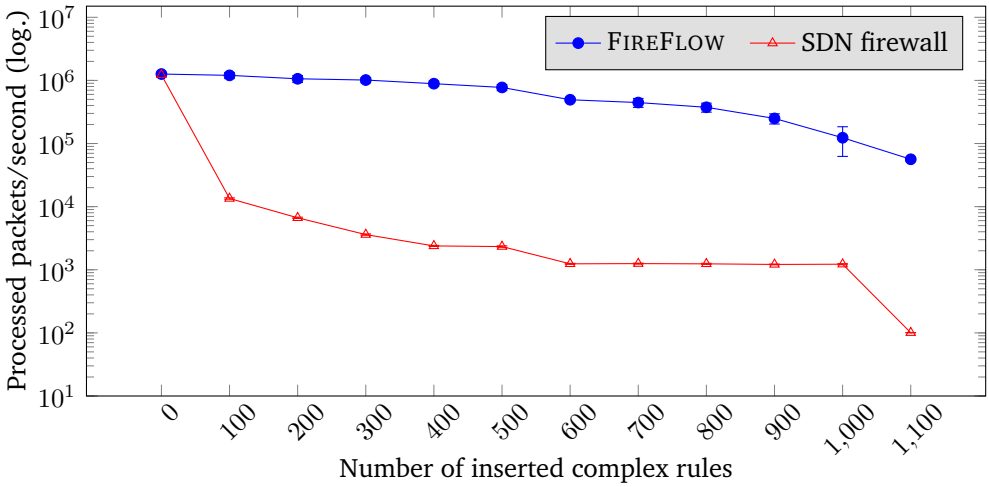


(a) Real rule sets.

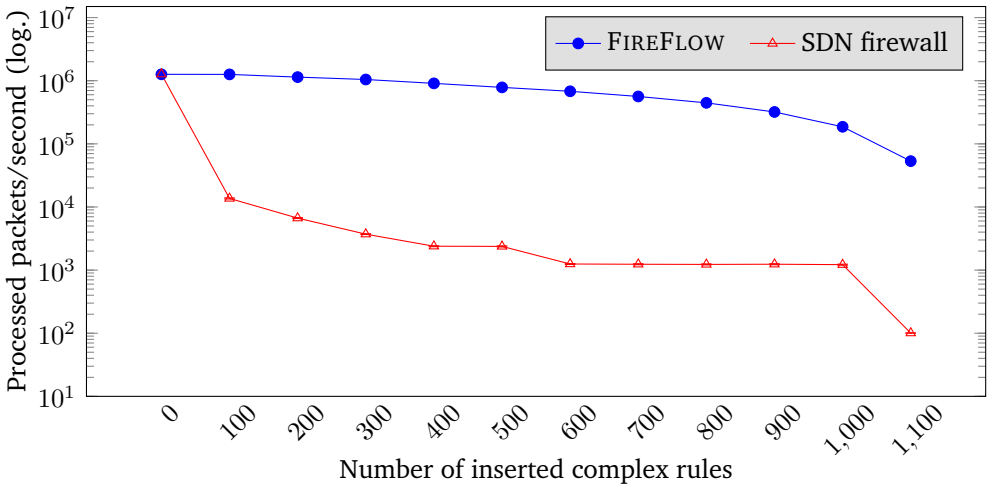


(b) Synthetic rule sets.

Fig. 6.9: Throughput comparison of FireFlow and stand-alone software firewall (`netfilter`) for varying numbers of complex rules. The data points show averaged throughputs for three real rule sets in 6.9a, and ten synthetic rule sets in 6.9b. The error bars show the respective standard deviations.



(a) Real rule sets.



(b) Synthetic rule sets.

Fig. 6.10: Throughput comparison of FireFlow and a plain SDN switch/firewall application setup for varying numbers of complex rules. The data points show averaged throughputs for three real rule sets in 6.10a, and ten synthetic rule sets in 6.10b. The error bars show the respective standard deviations.

rule part, this is not the case for the plain SDN setup, since the SDN controller software does not implement such algorithms. Instead, packets are sent back to the SDN switch directly. This can be seen as a best-case measurement for the SDN reference, as only the overhead of sending packets to the control plane and back is involved. For the evaluation results this means the performance-increasing effect of FireFlow would be even greater if the complex processing at the SDN application is taken into account, too. The negative impact due to the diversion to the control plane is still clearly visible.

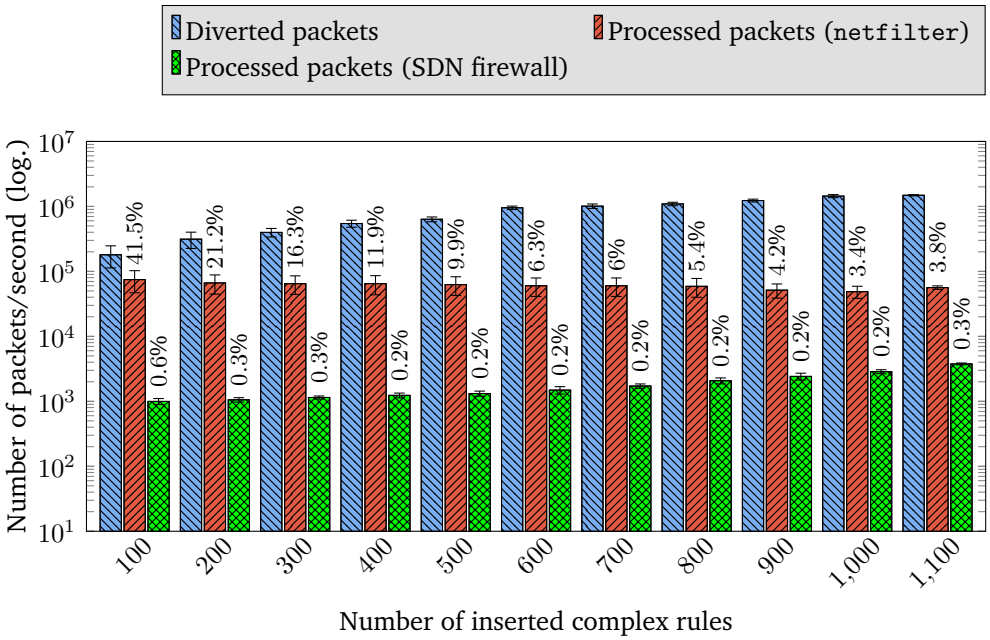
As expected, the second reference using a native SDN firewall setup severely suffered from the slow control channel as soon as complex rules were inserted. This can be seen in Figures 6.10a and 6.10b. In comparison, FIREFLOW processed about 500 times more packets.

6.5.5 Processing of Diverted Packets

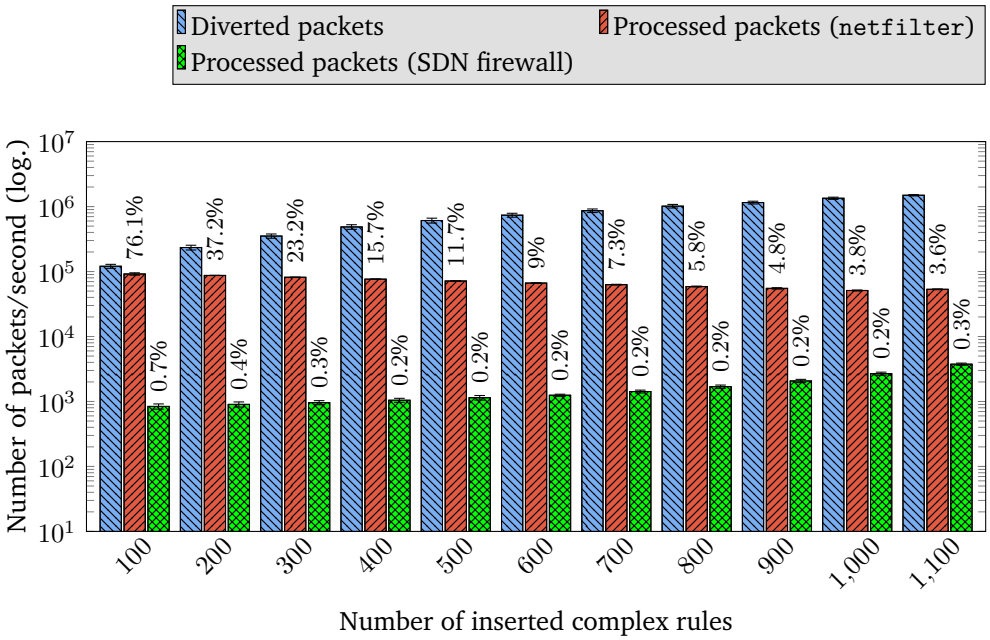
Another interesting parameter is the number of packets that are lost due to the diversion operation to the slower software firewall or the SDN controller, respectively. Figures 6.11a and 6.11b show a comparison between the number of packets that are diverted and those that are successfully processed by the back-end (either the software firewall or the SDN application). Even with few complex rules, most of the packets sent to the SDN application are lost, which means connections targeting complex rules will suffer immense packet loss. The software firewall back-end of FIREFLOW can sustain the workload better, especially if only few complex rules are present.

6.5.6 Update Latency

For operational purposes, the update latency for deploying rule set updates is also of interest. The update process involves several steps. First, the rule set must be configured via the management tool. For each update, the header space dependencies must be regarded and the corresponding subsets for the SDN switch and the software firewall must be calculated. The management tool then updates the SDN switch in our setup via REST API, as well as the software firewall with the standard `iptables-restore` command. We measured the time required for this process for an increasing number of complex rules. Figure 6.12 shows the latency and how it is affected by the number of rules. The computation effort for the HSA scales with $O(n^2)$ and therefore increases with more complex rules. However, in our case we can reduce the computation effort, since already identified HSA-dependent rules can be omitted for analysis. Furthermore, the number of rules that must be placed in the switch is decreasing with a larger fraction of complex rules. Since the REST API calls placing those rules consume a significant share of the update latency, this results in a decreasing update latency after a certain fraction is reached.



(a) Real rule sets.



(b) Synthetic rule sets.

Fig. 6.11: Average number of diverted packets and share of the diverted packets that could be processed by the SDN controller and firewall application (*SDN firewall*) compared with the FIREFLOW software firewall compartment (*netfilter*). The error bars show standard deviations. Percentages indicate the fraction of processed vs. diverted packets, the remainder are lost packets.

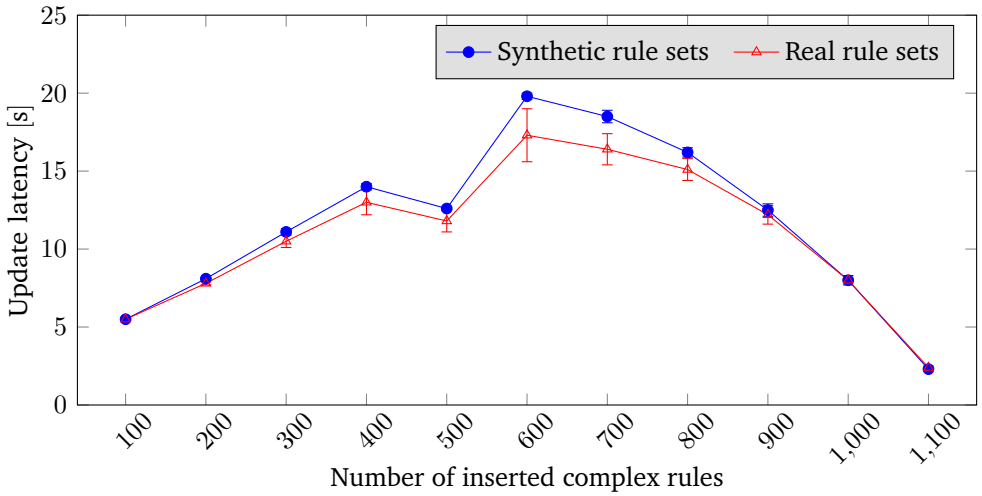


Fig. 6.12: Average time required to update the rule set in the evaluation FIREFLOW setup, depending on the number of complex rules. The error bars show standard deviations.

6.6 Summary

Building a powerful SDN-based firewall setup bears several challenges, which we could identify in this chapter. FIREFLOW was introduced as a measure to set up a fast and flexible hybrid SDN firewall setup. The concept primarily relies on the fast SDN data plane to process the majority of the traffic. Only decisions too complex to be processed within the SDN switch are diverted to a powerful software firewall within the context of the firewall's data plane. By applying an algorithm based on the already previously successfully utilized HSA, we can ensure that all decisions are still correct after the separation onto two different classification compartments with respect to the master rule set defined by the administrator. Still, we do not need to place the entire rule set onto the software firewall which again improves performance. Only standard components are used—in our example setup a COTS SDN switch and a software firewall based on `netfilter/iptables`. The evaluation shows that significant classification performance gains can be achieved. The approach further enables portability to other technologies, e.g., P4 hardware instead of an SDN switch, where additional features could be exploited.

7 Conclusion

In this thesis, we discussed design approaches for hybrid network packet processing systems, with an emphasis on network security appliances. Fundamental challenges in setup, operation, and administration could be identified for the proposed structure of combining high-performance networking hardware with the comprehensive analysis capabilities of software firewalls. This work contributes to the applicability of such hybrid designs by describing practical approaches that can exploit the strengths of hybrid network packet processing systems.

The basic challenges of network packet classification were described in Chapter 2. The chapter continued with an analysis of existing approaches used for network packet processing and classification, reaching from software-based systems to dedicated network processing hardware. The advantages and weaknesses of these approaches were discussed, which further justify the effort for building hybrid systems.

In Chapter 3, the first hybrid approach HyPaFilter—exploiting a standard software firewall in combination with a newly developed FPGA classification circuit—was introduced. Here, an FPGA serves as the primary classification system, diverting only packets to the software firewall where a complex decision (i.e., a complex rule) must be analysed. This concept demonstrated the general feasibility of hybrid network packet processing using these technologies as compartments. We further identified challenges occurring with this approach, in particular, dependencies among the rules of the rule set. The flexibility of a custom hardware circuit allowed us to exploit further possibilities for improvement. Most notably, the usage of matching meta information from the FPGA's classification circuit could be used to speed-up the subsequent software classification process. This enables an up to 30-fold performance increase compared to a standard software firewall. The evaluation confirms the approach as an answer to the first research question of this thesis: *How can we realize a combination of two packet classification systems so that each compartment processes the tasks it is best suited for in order to achieve high throughput, flexibility and comprehensive analysis capabilities?*

Chapter 4 expands Chapter 3. While the former approach primarily optimized the software classification process, the focus was now shifted towards better avoidance of the expensive diversion to the software firewall. To this end, the decision whether a packet needs to be diverted was determined by an algorithm based on header space analysis (HSA), which uses a geometric representation of the rules in order to find dependencies. One remarkable result of the evaluation is that neither the synthetic nor the real rule sets were subject to exceeding dependencies amongst the rules. Consequently, the implementation of the HSA dependency analysis allowed major parts of the traffic to be handled solely by the FPGA, even for more complex and demanding scenarios than those we used during the evaluation in Chapter 3. This allows the hybrid firewall to sustain the performance increase even with many and scattered complex rules.

With the goal of delegating more network-related processing natively to the FPGA, Chapter 5 described potential suitable network processing tasks and their application. Afterwards, two important applications were identified and analysed. First, an approach for resource-efficient, hybrid on-chip classification circuits that can be updated dynamically was introduced. This approach was evaluated and compared against standard hardware classification circuits, where it showed significantly reduced resource and power consumption. Secondly, hash functions, key building blocks for numerous network processing applications, are evaluated on their implementability on FPGAs. It could be shown that the recent hash function SHA3—while exceeding the requirements for this task—is better suited for FPGAs than weaker algorithms that are optimized for CPU-based implementation. Both findings support the proposition for further integration of networking tasks into FPGAs, hereby contributing to answering the second research question: *Which network packet processing tasks are desirable and feasible for FPGA implementation and what implementation approaches and optimizations are possible?*

Finally, the portability of the hybrid approach was demonstrated in Chapter 6, where the former FPGA-based hardware classification system was replaced by a commodity SDN hardware switch. Although the limitation of using standard hardware has constraints with regard to possible performance-enhancing extensions, a significant increase of the classification rate could be shown. Furthermore, FIREFLOW allows for an easy extension and integration into existing network infrastructure. The results answer the third research question, since standard SDN components without further modification are successfully used: *How can the hybrid classification approach be applied to other types of packet classification systems if one goal is to avoid modifications to the components themselves?* Table 7.1 shows how the contributed approaches compare against the state-of-the-art that was described in Chapter 2.

Approach	COTS	Open Src.	Flexibility	Analysis Features	HW-Offl.	Perf.
Linux netfilter	Yes	Yes	High	Comprehensive	None	Limited
Linux eBPF	Yes	Yes	High	Comprehensive	None	Limited
DPDK/pfSense [63]	Yes	Yes	Limited	Limited	None	Up to line speed
The shunt [92]	Partly[1]	Partly[2]	Medium	Comprehensive	Flows	Up to line speed
NFShunt [62]	Yes	Yes	Medium	Comprehensive	Flows	Up to line speed
FPGA stand-alone	Partly	Partly[2]	Medium	Limited	Full	Up to line speed
Com. HW firewall	No[3]	No	Limited	Varies	Full	Up to line speed
HyPaFilter+	Partly	Partly[2]	High	Comprehensive	Policies	Up to line speed
FIREFLOW	Yes	Yes	High	Comprehensive	Policies	Up to line speed

Tab. 7.1: Qualitative comparison of different packet classification approaches for firewalling, including the contributed ones.

[1]Modifications required to standard software components.
[2]Design contains closed-source components.
[3]Vendor-specific, not interchangeable.

The introduced approaches are designed to allow for flexible extensions. This way, technologies like stateful packet filtering can be integrated into the hybrid system. Moreover, a porting to other platforms, like NPUs or programmable network hardware based on P4, seems feasible.

Summarizing the results, this thesis provides a novel hybrid packet processing approach that can be utilized in different setups and environments. The approach is thoroughly evaluated, also giving an outlook to further extensions and portability.

Bibliography

Publications by the Author

[1] A. Fiessler, S. Hager, and B. Scheuermann. „Flexible line speed network packet classification using hybrid on-chip matching circuits“. In: *HPSR '17: Proceedings of the 2017 IEEE 18th International Conference on High Performance Switching and Routing*. Campinas, Brazil, June 2017, pp. 1–8 (cit. on pp. 23, 73, 77).

[2] A. Fiessler, S. Hager, B. Scheuermann, and A.W. Moore. „HyPaFilter - A Versatile Hybrid FPGA Packet Filter“. In: *ANCS '16: Proceedings of the 2016 ACM/IEEE Ninth Symposium on Architectures for Networking and Communication Systems*. Santa Clara, CA, USA, Mar. 2016, pp. 25–36 (cit. on pp. 19, 24, 49).

[3] A. Fiessler, D. Loebenberger, S. Hager, and B. Scheuermann. „On the Use of (Non-)Cryptographic Hashes on FPGAs“. In: *Applied Reconfigurable Computing*. Springer International Publishing, 2017, pp. 72–80. ISBN: 978-3-319-56258-2 (cit. on p. 73).

[4] A. Fiessler, C. Lorenz, S. Hager, and B. Scheuermann. „FireFlow – High Performance Hybrid SDN-Firewalls with OpenFlow“. In: *LCN '18: Proceedings of the 2018 IEEE 43rd Conference on Local Computer Networks*. Chicago, IL, USA, Oct. 2018, pp. 227–230 (cit. on p. 99).

[5] A. Fiessler, C. Lorenz, S. Hager, B. Scheuermann, and A. W. Moore. „HyPaFilter+: Enhanced Hybrid Packet Filtering Using Hardware Assisted Classification and Header Space Analysis“. In: *IEEE/ACM Transactions on Networking* 25.6 (Dec. 2017), pp. 3655–3669. ISSN: 1063-6692 (cit. on pp. 19, 49, 74, 106).

[6] S. Hager, P. John, A. Fiessler, and B. Scheuermann. „Minflate: Combining Rule Set Minimization with Jump-based Expansion for Fast Packet Classification“. In: *ANCS '16: Proceedings of the 2016 ACM/IEEE Ninth Symposium on Architectures for Networking and Communication Systems*. Santa Clara, CA, USA, Mar. 2016, pp. 115–116 (cit. on p. 11).

Other Publications

[7] K. Accardi, T. Bock, F. Hady, and J. Krueger. „Network processor acceleration for a Linux* netfilter firewall“. In: *ANCS '05: Proceedings of the 2005 Symposium on Architectures for Networking and Communication Systems*. Princeton, New Jersey, USA, Oct. 2005, pp. 115–123 (cit. on pp. 17, 21, 22, 101).

[8] M. Attig, S. Dharmapurikar, and J. Lockwood. „Implementation results of bloom filters for string matching“. In: *FCCM '04: Proceedings of the 2004 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa, CA, USA, Apr. 2004, pp. 322–323 (cit. on p. 88).

[9] J. Aumasson and D. J. Bernstein. „SipHash: A Fast Short-Input PRF“. In: *Progress in Cryptology - INDOCRYPT 2012*. Springer Berlin Heidelberg, 2012, pp. 489–508. ISBN: 978-3-642-34931-7 (cit. on pp. 90, 94).

[10] F. Baboescu and G. Varghese. „Scalable Packet Classification“. In: *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. San Diego, CA, USA, Aug. 2001, pp. 199–210 (cit. on pp. 11, 74).

[11] Z. K. Baker and V. K. Prasanna. „Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs“. In: *Field Programmable Logic and Application*. Springer Berlin Heidelberg, 2004, pp. 311–321. ISBN: 978-3-540-30117-2 (cit. on p. 16).

[12] N. Bar-Yosef and A. Wool. „Remote Algorithmic Complexity Attacks against Randomized Hash Tables“. In: *E-business and Telecommunications*. Springer Berlin Heidelberg, 2009, pp. 162–174. ISBN: 978-3-540-88653-2 (cit. on p. 88).

[13] E. Blanton and M. Allman. „On making TCP more robust to packet reordering“. In: *ACM SIGCOMM Computer Communication Review* 32.1 (2002), pp. 20–30. ISSN: 0146-4833 (cit. on p. 33).

[14] B. H. Bloom. „Space/time trade-offs in hash coding with allowable errors“. In: *Communications of the ACM* 13.7 (1970), pp. 422–426. ISSN: 0001-0782 (cit. on pp. 86, 88, 89).

[15] A. Bookstein. „Double hashing“. In: *Journal of the American Society for Information Science* 23.6 (1972), pp. 402–405 (cit. on pp. 86, 89).

[16] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. „P4: Programming Protocol-independent Packet Processors“. In: *ACM SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833 (cit. on pp. 14, 101).

[17] A. Broder and M. Mitzenmacher. „Network applications of bloom filters: A survey“. In: *Internet mathematics* 1.4 (2004), pp. 485–509 (cit. on p. 88).

[18] A. Broder and M. Mitzenmacher. „Using multiple hash functions to improve IP lookups“. In: *INFOCOM '01: Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies.* Vol. 3. Anchorage, AK, USA, 2001 (cit. on pp. 86, 87).

[19] Y. Chang. „A 2-Level TCAM Architecture for Ranges“. In: *IEEE Transactions on Computers* 55.12 (Dec. 2006), pp. 1614–1629. ISSN: 0018-9340 (cit. on pp. 12, 13).

[20] H. Chen, Y. Chen, and D.H. Summerville. „A Survey on the Application of FPGAs for Network Infrastructure Security“. In: *Communications Surveys & Tutorials, IEEE* 13.4 (2011), pp. 541–561. ISSN: 1553-877X (cit. on pp. 2, 16).

[21] M. Chen, M. Liao, P. Tsai, M. Luo, C. Yang, and C. E. Yeh. „Using NetFPGA to Offload Linux Netfilter Firewall“. In: *2nd North American NetFPGA Developers Workshop*. Stanford, CA, USA, Aug. 2010 (cit. on pp. 17, 22).

[22] C. R. Clark and D. E. Schimmel. „Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns“. In: *Field Programmable Logic and Application*. Springer Berlin Heidelberg, 2003, pp. 956–959. ISBN: 978-3-540-45234-8 (cit. on pp. 14, 16).

[23] J. Collings and J. Liu. „An OpenFlow-Based Prototype of SDN-Oriented Stateful Hardware Firewalls“. In: *ICNP ’14: Proceedings of the 2014 IEEE 22nd International Conference on Network Protocols*. Raleigh, NC, USA, Oct. 2014, pp. 525–528 (cit. on p. 102).

[24] S. Crosby and D. Wallach. „Denial of Service via Algorithmic Complexity Attacks.“ In: *Proceedings of the 12th USENIX Security Symposium*. Washington, DC, USA, Aug. 2003, pp. 29–44 (cit. on pp. 74, 88, 89).

[25] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary. „An FPGA-Based Network Intrusion Detection Architecture“. In: *IEEE Transactions on Information Forensics and Security* 3.1 (Mar. 2008), pp. 118–132. ISSN: 1556-6013 (cit. on pp. 2, 90).

[26] I. Dubrawsky. „Firewall evolution-deep packet inspection“. In: *Security Focus* 29 (2003) (cit. on pp. 2, 8, 99).

[27] R. Durner and W. Kellerer. „The cost of Security in the SDN control Plane“. In: *ACM CoNEXT Student Workshop ’15*. Heidelberg, Germany, Dec. 2015 (cit. on pp. 100, 103, 104).

[28] M. Dworkin. *FIPS PUB 202. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Aug. 2015 (cit. on pp. 90, 94).

[29] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. „MoonGen: A Scriptable High-Speed Packet Generator“. In: *IMC ’15: Proceedings of the 2015 Internet Measurement Conference*. Tokyo, Japan, Oct. 2015, pp. 275–287. ISBN: 978-1-4503-3848-6 (cit. on p. 11).

[30] H. Feistel. „Cryptography and Computer Privacy“. In: *Scientific American* 228.5 (1973), pp. 15–23. ISSN: 00368733, 19467087 (cit. on pp. 86, 90).

[31] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang. „ParaSplit: A Scalable Architecture on FPGA for Terabit Packet Classification“. In: *HOTI ’12: Proceedings of the 20th Symposium on High Performance Interconnects*. Santa Clara, CA, USA, Aug. 2012, pp. 1–8 (cit. on pp. 16, 105).

[32] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. „Comparison of Frameworks for High-Performance Packet IO“. In: *ANCS '15: Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. Oakland, California, USA, May 2015, pp. 29–38. ISBN: 978-1-4673-6632-8 (cit. on p. 11).

[33] T. Ganegedara and V. Prasanna. „StrideBV: Single Chip 400G+ Packet Classification“. In: *HPSR'12: IEEE 13th Conference on High Performance Switching and Routing*. Belgrade, Serbia, June 2012, pp. 1–6 (cit. on pp. 23, 75).

[34] K. Golnabi, R. K. Min, L. Khan, and E. Al-Shaer. „Analysis of Firewall Policy Rules Using Data Mining Techniques“. In: *NOMS '06: Proceedings of the 2006 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2006, pp. 305–315 (cit. on pp. 49, 74).

[35] P. Gupta and N. McKeown. „Algorithms for Packet Classification“. In: *IEEE Network: The Magazine of Global Internetworking* 15.2 (Mar. 2001), pp. 24–32. ISSN: 0890-8044 (cit. on pp. 7, 11).

[36] P. Gupta and N. McKeown. „Packet Classification using Hierarchical Intelligent Cuttings“. In: *HOTI '99: Proceedings of the 7th Symposium on High Performance Interconnects*. Stanford, CA, USA, Aug. 1999, pp. 34–41 (cit. on pp. 11, 74).

[37] S. Hager, D. Bendyk, and B. Scheuermann. „Partial Reconfiguration And Specialized Circuitry for Flexible FPGA-based Packet Processing“. In: *ReConFig '15: 2015 International Conference on ReConFigurable Computing and FPGAs*. Mayan Riviera, Mexico, Dec. 2015, pp. 1–6 (cit. on pp. 23, 76).

[38] S. Hager, S. Selent, and B. Scheuermann. „Trees in the List: Accelerating List-based Packet Classification Through Controlled Rule Set Expansion“. In: *CoNEXT '14: Proceedings of the 10th International Conference on Emerging Networking Experiments and Technologies*. Sydney, Australia, Dec. 2014, pp. 101–107 (cit. on pp. 11, 21, 30, 50).

[39] S. Hager, F. Winkler, B. Scheuermann, and K. Reinhardt. „MPFC: Massively Parallel Firewall Circuits“. In: *LCN '14: Proceedings of the 39th Annual IEEE International Conference on Local Computer Networks*. Edmonton, Canada, Sept. 2014, pp. 305–313 (cit. on pp. 16, 22, 76, 105).

[40] F. Heimgaertner, M. Schmidt, D. Morgenstern, and M. Menth. „A Software-Defined Firewall Bypass for Congestion Offloading“. In: *CNSM '17: Proceedings of the 2017 13th International Conference on Network and Service Management*. Tokyo, Japan, Nov. 2017, pp. 1–9 (cit. on p. 103).

[41] H. Hu, W. Han, G. Ahn, and Z. Zhao. „FlowGuard: Building Robust Firewalls for Software-defined Networks“. In: *HotSDN '14: Proceedings of the third workshop on Hot topics in software defined networking*. Chicago, IL, USA, 2014, pp. 97–102 (cit. on pp. 99, 102).

[42] „IEEE Standard for Verilog Hardware Description Language“. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–560 (cit. on p. 15).

[43] „IEEE Standards Interpretations: IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual“. In: *IEEE Std 1076/INT-1991* (1992) (cit. on p. 15).

[44] Cisco Visual Networking Index. *The zettabyte era–trends and analysis*. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf. Last access: Nov 2018. 2016 (cit. on p. 1).

[45] Intel Corporation. *DPDK Intel NIC Performance Report Release 17.08*. http://fast.dpdk.org/doc/perf/DPDK_17_08_Intel_NIC_performance_report.pdf. Last access: Dec 10, 2017. 2017 (cit. on p. 11).

[46] B. Jenkins. *Various publications on hash functions*. http://www.burtleburtle.net/bob/hash/doobs.html and http://www.burtleburtle.net/bob/hash/spooky.html and http://www.burtleburtle.net/c/lookup2.c and http://www.burtleburtle.net/bob/c/lookup3.c. Last access: Nov 15, 2016 (cit. on pp. 90, 94).

[47] J. Jeong, J. Seo, G. Cho, H. Kim, and J. Park. „A Framework for Security Services Based on Software-Defined Networking“. In: *WAINA '15: Proceedings of the 2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*. Gwangiu, South Korea, Mar. 2015, pp. 150–153 (cit. on p. 102).

[48] W. Jiang and V. Prasanna. „A FPGA-based Parallel Architecture for Scalable High-Speed Packet Classification“. In: *ASAP '09: Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*. Boston, MA, USA, July 2009, pp. 24–31 (cit. on pp. 14, 16, 105).

[49] W. Jiang and V. Prasanna. „Large-scale Wire-speed Packet Classification on FPGAs“. In: *FPGA '09: Proceedings of the ACM/SIGDA 17th International Symposium on Field Programmable Gate Arrays*. Monterey, California, USA, Feb. 2009, pp. 219–228 (cit. on pp. 1, 2, 10, 14, 16, 73, 75, 105).

[50] P. Kazemian, G. Varghese, and N. McKeown. „Header Space Analysis: Static Checking for Networks“. In: *NSDI '12: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*. San Jose, CA, USA, Apr. 2012, pp. 113–126 (cit. on p. 49).

[51] S. Kilts. *Advanced FPGA design: architecture, implementation, and optimization*. John Wiley & Sons, 2007. ISBN: 9780470127896 (cit. on p. 15).

[52] V. Klima. „Tunnels in Hash Functions: MD5 Collisions Within a Minute.“ In: *IACR Cryptology ePrint Archive* 2006 (2006), p. 105. URL: https://eprint.iacr.org/2006/105.pdf (cit. on p. 88).

[53] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. „Software-Defined Networking: A Comprehensive Survey“. In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. ISSN: 0018-9219. URL: http://arxiv.org/abs/1406.0440 (cit. on pp. 13, 14, 99, 104).

[54] T. Lakshman and D. Stiliadis. „High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching“. In: *SIGCOMM '98: Proceedings of the 1998 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Vancouver, BC, Canada, Aug. 1998, pp. 203–214 (cit. on pp. 11, 50, 74).

[55] A. Liu and M. Gouda. „Complete Redundancy Detection in Firewalls“. In: *Data and Applications Security XIX*. Storrs, CT, USA: Springer Berlin Heidelberg, Aug. 2005, pp. 193–206. ISBN: 978-3-540-31937-5 (cit. on p. 108).

[56] A. Liu, E. Torng, and C. Meiners. „Firewall Compressor: An Algorithm for Minimizing Firewall Policies“. In: *INFOCOM '08: Proceedings of the 27th Annual Joint Conference of the IEEE Computer and Communications Societies*. Phoenix, AZ, USA, Apr. 2008, pp. 176–180 (cit. on pp. 11, 21).

[57] D. Liu, B. Hua, X. Hu, and X. Tang. „High-performance Packet Classification Algorithm for Many-core and Multithreaded Network Processor“. In: *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded System*. Seoul, Korea, Oct. 2006, pp. 334–344 (cit. on p. 14).

[58] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. „NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing“. In: *MSE '07: Proceedings of the International Conference on Microelectronic Systems Education*. San Diego, CA, USA, June 2007, pp. 160–161 (cit. on p. 34).

[59] J. W Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers. „A low-latency library in FPGA hardware for high-frequency trading (HFT)“. In: *HOTI '12: Proceedings of the 20th Annual Symposium on High-Performance Interconnects*. IEEE. Santa Clara, CA, USA, Aug. 2012, pp. 9–16 (cit. on pp. 2, 16).

[60] C. Lorenz, D. Hock, J. Scherer, R. Durner, W. Kellerer, S. Gebert, N. Gray, T. Zinner, and P. Tran-Gia. „An SDN/NFV-Enabled Enterprise Network Architecture Offering Fine-Grained Security Policy Enforcement“. In: *IEEE Communications Magazine* 55.3 (Mar. 2017), pp. 217–223 (cit. on p. 103).

[61] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. „OpenFlow: Enabling Innovation in Campus Networks“. In: *ACM SIGCOMM Computer Communication Review* 38.2 (Mar. 2008), pp. 69–74 (cit. on pp. 14, 104).

[62] S. Miteff and S. Hazelhurst. „NFShunt: A Linux firewall with OpenFlow-enabled hardware bypass“. In: *NFV-SDN '15*. San Francisco, CA, USA, Nov. 2015, pp. 100–106 (cit. on pp. 18, 103, 121).

[63] Netgate. *Further (a roadmap for pfSense)*. https://www.netgate.com/blog/further-a-roadmap-for-pfsense.html. Last access: Dec 06, 2017. 2015 (cit. on pp. 11, 18, 121).

[64] B. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti. „A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks“. In: *IEEE Communications Surveys Tutorials* 16.3 (Feb. 2014), pp. 1617–1634 (cit. on pp. 14, 100).

[65] D. H. Nyang. *Counting bloom filter*. US Patent 9,740,797. Aug. 2017 (cit. on p. 88).

[66] R. Pagh and F. Rodler. „Cuckoo hashing“. In: *European Symposium on Algorithms*. Springer. 2001, pp. 121–133 (cit. on pp. 86, 89).

[67] J. Pena and W. Yu. „Development of a Distributed Firewall Using Software Defined Networking Technology“. In: *ICIST '14*. Apr. 2014, pp. 449–452 (cit. on p. 102).

[68] B. Pfaff, J. Scherer, D. Hock, N. Gray, T. Zinner, P. Tran-Gia, R. Durner, W. Kellerer, and C. Lorenz. „SDN/NFV-enabled Security Architecture for Fine-grained Policy Enforcement and Threat Mitigation for Enterprise Networks“. In: *SIGCOMM '17*. Los Angeles, CA, USA, Aug. 2017, pp. 15–16 (cit. on p. 103).

[69] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, et al. „A reconfigurable fabric for accelerating large-scale datacenter services“. In: *ISCA '14: Proceedings of the 41st International Symposium on Computer Architecture*. Minneapolis, MA, USA, June 2014, pp. 13–24 (cit. on p. 20).

[70] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li. „Towards High-performance Flow-level Packet Processing on Multi-core Network Processors“. In: *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architectures for Networking and Communication Systems*. Orlando, Florida, USA, Dec. 2007, pp. 17–26 (cit. on p. 14).

[71] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. „Packet Classification Algorithms: From Theory to Practice“. In: *INFOCOM '09: Proceedings of the 28th Annual Joint Conference of the IEEE Computer and Communications Societies*. Rio de Janeiro, Brazil, Apr. 2009, pp. 648–656 (cit. on p. 74).

[72] Y. Qu and V. Prasanna. „High-Performance and Dynamically Updatable Packet Classification Engine on FPGA“. In: *IEEE Transactions on Parallel and Distributed Systems* 27.1 (Jan. 2016), pp. 197–209. ISSN: 1045-9219 (cit. on p. 75).

[73] D. Qunfeng, S. Banerjee, J. Wang, and D. Agrawal. „Wire Speed Packet Classification Without TCAMs: A Few More Registers (And A Bit of Logic) Are Enough“. In: *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. San Diego, CA, USA, June 2007, pp. 253–264 (cit. on pp. 12, 23, 74, 75).

[74] R. Ramanathan and J. Redi. „A brief overview of ad hoc networks: challenges and directions“. In: *IEEE communications Magazine* 40.5 (2002), pp. 20–22. ISSN: 0163-6804 (cit. on p. 9).

[75] L. Rizzo. „Netmap: A Novel Framework for Fast Packet I/O“. In: *USENIX ATC '12: Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. Boston, MA, 2012, p. 9 (cit. on p. 11).

[76] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy. „Exact Worst Case TCAM Rule Expansion“. In: *IEEE Transactions on Computers* 62.6 (June 2013), pp. 1127–1140. ISSN: 0018-9340 (cit. on p. 12).

[77] R. Sangireddy and A. Somani. „High-Speed IP Routing With Binary Decision Diagrams Based Hardware Address Lookup Engine“. In: *IEEE Journal on Selected Areas in Communications* 21.4 (May 2003). ISSN: 0733-8716 (cit. on p. 76).

[78] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle. „Performance Implications of Packet Filtering with Linux eBPF“. In: *ITC 30: Proceedings of the 2018 30th International Teletraffic Congress*. Vol. 01. Vienna, Austria, Sept. 2018, pp. 209–217 (cit. on p. 12).

[79] D. V. Schuehler and J. W. Lockwood. „A Modular System for FPGA-Based TCP Flow Processing in High-Speed Networks“. In: *Field Programmable Logic and Application*. Springer Berlin Heidelberg, 2004, pp. 301–310. ISBN: 978-3-540-30117-2 (cit. on pp. 14, 16).

[80] E. S. Al-Shaer and H. H. Hamed. „Discovery of policy anomalies in distributed firewalls“. In: *INFOCOM '04: Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communication Societies*. Vol. 4. Hong Kong, China, Mar. 2004, 2605–2616 vol.4 (cit. on p. 17).

[81] Z. Shi, C. Ma, J. Cote, and B. Wang. „Hardware implementation of hash functions“. In: *Introduction to Hardware Security and Trust*. Springer, 2012, pp. 27–50. ISBN: 978-1-4419-8080-9 (cit. on pp. 74, 86–88).

[82] S. Singh, F. Baboescu, G. Varghese, and J. Wang. „Packet Classification Using Multidimensional Cutting“. In: *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Karlsruhe, Germany, Aug. 2003, pp. 213–224 (cit. on p. 11).

[83] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. „Fast hash table lookup using extended bloom filter: an aid to network processing“. In: *ACM SIGCOMM Computer Communication Review* 35.4 (2005), pp. 181–192. ISSN: 0146-4833 (cit. on pp. 87, 88, 90, 95).

[84] H. Song and J. W. Lockwood. „Efficient Packet Classification for Network Intrusion Detection using FPGA“. In: *FPGA '05: Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*. Monterey, California, USA, Feb. 2005, pp. 238–245 (cit. on pp. 10, 14, 16).

[85] I. Sourdis and D. Pnevmatikatos. „Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System“. In: *Field Programmable Logic and Application*. Springer Berlin Heidelberg, 2003, pp. 880–889. ISBN: 978-3-540-45234-8 (cit. on p. 16).

[86] E. Spitznagel, D. Taylor, and J. Turner. „Packet Classification Using Extended TCAMs“. In: *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*. Atlanta, Georgia, USA, Nov. 2003, pp. 120–131 (cit. on p. 12).

[87] V. Srinivasan, S. Suri, and G. Varghese. „Packet Classification using Tuple Space Search“. In: *SIGCOMM '99: Proceedings of the 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Cambridge, MA, USA, Aug. 1999, pp. 135–146 (cit. on p. 11).

[88] D. Taylor. „Survey and Taxonomy of Packet Classification Techniques“. In: *ACM Comput. Surv.* 37.3 (Sept. 2005), pp. 238–275. ISSN: 0360-0300 (cit. on p. 7).

[89] D. Taylor and J. Turner. „ClassBench: a packet classification benchmark“. In: *IEEE/ACM Transactions on Networking* 15.3 (June 2007), pp. 499–511. ISSN: 1063-6692 (cit. on pp. 36, 37, 78, 110).

[90] K. Thompson, G. J. Miller, and R. Wilder. „Wide-area Internet traffic patterns and characteristics“. In: *IEEE network* 11.6 (1997), pp. 10–23. ISSN: 0890-8044 (cit. on p. 7).

[91] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. „Multi-Layer Packet Classification with Graphics Processing Units“. In: *CoNEXT '14: Proceedings of the 10th International Conference on Emerging Networking Experiments and Technologies*. Sydney, Australia, Dec. 2014, pp. 109–120 (cit. on p. 14).

[92] N. Weaver, V. Paxson, and J.M Gonzalez. „The shunt: an FPGA-based accelerator for network intrusion prevention“. In: *FPGA '07: Proceedings of the ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. Monterey, California, USA, Feb. 2007, pp. 199–206 (cit. on pp. 17, 18, 20, 121).

[93] F. Yu and R. H. Katz. „Efficient Multi-Match Packet Classification with TCAM“. In: *HOTI '04: Proceedings of the 12th Symposium on High Performance Interconnects*. Stanford, CA, USA, Aug. 2004, pp. 28–34 (cit. on pp. 12, 13).

[94] L. Yuan, H. Chen, J. Mai, C. Chuah, Z. Su, and P. Mohapatra. „FIREMAN: a toolkit for firewall modeling and analysis“. In: *S&P '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*. Berkeley/Oakland, CA, USA, May 2006, pp. 15– (cit. on p. 8).

[95] S.s Zerkane, D. Espes, P. L. Parc, and F. Cuppens. „A Proactive Stateful Firewall for Software Defined Networking“. In: *Risks and Security of Internet and Systems*. Springer International Publishing, 2017, pp. 123–138. ISBN: 978-3-319-54876-0 (cit. on p. 102).

[96] N. Zilberman, Y. Audzevich, G.A. Covington, and A.W. Moore. „NetFPGA SUME: Toward 100 Gbps as Research Commodity“. In: *IEEE Micro* 34.5 (Sept. 2014), pp. 32–41. ISSN: 0272-1732 (cit. on pp. 16, 20, 32).

Online

[97] ARM. *AMBA AXI and ACE Protocol Specification*. Last access: Nov. 3, 2018. URL: `http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf` (cit. on p. 34).

[98] J. Aumasson and D. Bernstein. *C++ program to find universal (key-independent) multicollisions for CityHash64*. Last access: Nov. 3, 2018. URL: `https://131002.net/siphash/citycollisions-20120730.tar.gz` (cit. on pp. 90, 94).

[99] L. Byungjoon, J. Shin, and S. H. Park. *Openflow Controller by ETRI*. Last access: Nov. 3, 2018. 2015. URL: `https://github.com/openiris/IRIS` (cit. on p. 46).

[100] *ClassBench parameter files from 12 real filter sets*. Last access: Nov. 3, 2018. URL: `https://www.arl.wustl.edu/classbench/parameter_files.tar.gz` (cit. on p. 37).

[101] CompaniesHistory.com. *Xilinx, Inc. History*. Last access: Nov. 3, 2018. URL: `http://www.companieshistory.com/xilinx/` (cit. on p. 16).

[102] EE Times. *Design Criteria for Searching Databases: Part 1*. Last access: Nov. 3, 2018. URL: `https://www.eetimes.com/document.asp?doc_id=1202975` (cit. on p. 12).

[103] EE Times. *Xilinx UltraScale FPGA Offers 50 Million Equivalent ASIC Gates*. Last access: Nov. 3, 2018. URL: `https://www.eetimes.com/document.asp?doc_id=1320345` (cit. on p. 15).

[104] *Federal Register / Vol. 72, No. 212*. Last access: Nov 3, 2018. URL: `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf` (cit. on pp. 87, 88).

[105] *Floodlight - Firewall REST API*. Last access: Nov. 3, 2018. 2015. URL: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343614/Firewall+REST+API (cit. on pp. 99, 102).

[106] Fortinet. *Improving FortiGate performance with flow-based UTM scanning*. Last access: Nov. 3, 2018. 2015. URL: http://docs-legacy.fortinet.com/cb/html/index.html#page/FOS_Cookbook/UTM/cb_utm_av_flow-based_scanning.html (cit. on p. 19).

[107] *genugate firewall*. Last access: Nov. 3, 2018. URL: www.genua.de/en/solutions/high-resistance-firewall-genugate.html (cit. on pp. 2, 19).

[108] Hewlett Packard Enterprise Development LP. *Data Sheet Aruba DS 3810 Switch Series*. Last access: Nov. 3, 2018. URL: http://www.arubanetworks.com//assets/ds/DS_3810SwitchSeries.pdf (cit. on pp. 105, 111).

[109] H. Hsing. *SHA3 (KECCAK)*. Last access: Nov. 3, 2018. 2012. URL: http://opencores.org/project,sha3 (cit. on p. 95).

[110] IEEE. *200 Gb/s and 400 Gb/s Ethernet Task Force*. Last access: Nov. 3, 2018. 2016. URL: http://www.ieee802.org/3/bs/index.html (cit. on p. 1).

[111] *IPFW Firewall*. Last access: Nov. 3, 2018. URL: https://www.freebsd.org/cgi/man.cgi?ipfw (cit. on pp. 19, 104).

[112] Juniper Networks, Inc. *Configuring Sophos Antivirus Feature Profile*. Last access: Nov. 3, 2018. 2017. URL: http://www.juniper.net/documentation/en_US/junos/topics/example/utm-antivirus-sophos-configuring-feature-profile.html (cit. on p. 19).

[113] Juniper Networks, Inc. *SRX5400, SRX5600, and SRX5800 Services Gateways*. Last access: Nov. 3, 2018. 2017. URL: https://www.juniper.net/assets/us/en/local/pdf/datasheets/1000254-en.pdf (cit. on p. 19).

[114] Kevin Deierling, The Next Platform. *In Modern Datacenters, The Latency Tail Wags The Network Dog*. Last access: Nov. 3, 2018. URL: https://www.nextplatform.com/2018/03/27/in-modern-datacenters-the-latency-tail-wags-the-network-dog/ (cit. on p. 44).

[115] Neustar, Inc. *Choosing a Good Hash Function, Part 3*. Last access: Nov. 3, 2018. Feb. 2012. URL: https://research.neustar.biz/2012/02/02/choosing-a-good-hash-function-part-3/ (cit. on p. 86).

[116] Neutronome Systems Inc. *Agilio CX 2x40GbE SmartNIC*. Last access: Nov. 3, 2018. URL: https://www.netronome.com/media/documents/PB_Agilio_CX_2x40GbE.pdf (cit. on p. 14).

[117] oCERT.org. *#2011-003 multiple implementations denial-of-service via hash algorithm collision*. Last access: Nov. 3, 2018. 2011. URL: http://www.ocert.org/advisories/ocert-2011-003.html (cit. on pp. 74, 88).

[118] oCERT.org. *#2012-001 multiple implementations denial-of-service via MurmurHash algorithm collision*. Last access: Nov. 3, 2018. 2012. URL: http://www.ocert.org/advisories/ocert-2012-001.html (cit. on pp. 74, 86, 88, 90, 94).

[119] *OpenBSD Packet Filter*. Last access: Nov. 3, 2018. URL: http://www.openbsd.org/faq/pf/ (cit. on pp. 8, 11, 19, 104).

[120] Palo Alto Networks. *How to Implement and Test SSL Decryption*. Last access: Nov. 3, 2018. 2010. URL: https://live.paloaltonetworks.com/docs/DOC-1412 (cit. on p. 19).

[121] Quanta Computer Inc. *QuantaMesh 5000 Series BMS T5016-LB8D*. Last access: Nov. 3, 2018. 2015. URL: https://www.qct.io/zh-CN/product/index/Networking/Bare-Metal-Switch/Spine-Switch/QuantaMesh-BMS-T5016-LB8D (cit. on p. 46).

[122] Secworks Sweden AB. *SipHash Verilog*. Last access: Nov. 3, 2018. 2016. URL: https://github.com/secworks/siphash (cit. on p. 95).

[123] Jisoo Shin. *REST API List of OFMFirewall*. Last access: Nov 3, 2018. 2014. URL: https://github.com/openiris/IRIS/wiki/REST-API-List-of-OFMFirewall (cit. on p. 47).

[124] SourceTech411, James Allen. *Engineering Laws – Moore's, Rock's, Butter's and others*. Last access: Nov 3, 2018. URL: http://sourcetech411.com/2012/12/engineering-laws-moores-rocks-butters-and-others/ (cit. on p. 1).

[125] RYU Project Team. *Ryubook 1.0 - Firewall*. Last access: Nov. 2, 2018. 2014. URL: https://osrg.github.io/ryu-book/en/html/rest_firewall.html (cit. on p. 102).

[126] The Linux Foundation Project. *Data Plane Development Kit*. Last access: Nov. 3, 2018. 2017. URL: http://dpdk.org/ (cit. on p. 11).

[127] *The netfilter.org project*. Last access: Nov. 3, 2018. URL: www.netfilter.org (cit. on pp. 11, 19, 104).

[128] Ullrich, S. *The Semantic Gap*. Last access: Nov. 3, 2018. 2017. URL: https://noxxi.de/research/semantic-gap.html (cit. on p. 19).

[129] Xilinx Inc. *Vivado Design Suite*. Last access: Nov 3, 2018. URL: https://www.xilinx.com/products/design-tools/vivado.html (cit. on p. 32).

Nomenclature

H	Header field value
$\mathcal{H}$	Tuple of header field values H
D	Range of non-negative integers
$\mathcal{U}$	Set of all possible header values
R_i	Firewall rule with index/priority i
R^-	Simple reduction of a firewall rule R
R_S	Simple firewall rule
R_C	Complex firewall rule
$\mathcal{R}$	Prioritized list of rules R_i
$\mathcal{R_S}$	Set of simple rules R_S
$\mathcal{R_C}$	Set of complex rules R_C
G	Group of consecutive simple rules R_S
Q	Group of consecutive complex rules R_C
$\mathcal{Q}$	Set of complex rule groups Q
I	Integer interval of rule indices i
$\mathcal{I}$	Set of integer intervals I
C	Checks contained in any rule R
E	Complex checks in any rule R
P	Network packet with header values
A	Action of any rule R
$V_\mathcal{R}$	Shunting vector relative to rule set $\mathcal{R}$
M	Matching unit
Γ_i	Set of complex rules in $\mathcal{R}$ with higher priority than R_i
$G(R)$	Geometric representation of rule R

Erklärung

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben. Ich habe mich nicht anderwärts um einen Doktorgrad in dem Promotionsfach beworben und besitze keinen entsprechenden Doktorgrad. Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 42 am 11. Juli 2018, habe ich zur Kenntnis genommen.

Declaration: I declare that I have completed the thesis independently using only the aids and tools specified. I have not applied for a doctor's degree in the doctoral subject elsewhere and do not hold a corresponding doctor's degree. I have taken due note of the Faculty of Mathematics and Natural Sciences PhD Regulations, published in the Official Gazette of Humboldt Universität zu Berlin no. 42 on July 11 2018.